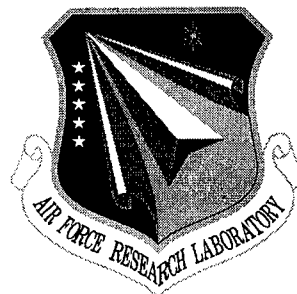


AFRL-IF-RS-TR-2001-172
Final Technical Report
August 2001



COMPOSABILITY FOR SECURE SYSTEMS

Secure Computing Corporation

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D849

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

©Copyright, 1998, Secure Computing Corporation. All Rights Reserved. This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227-7013 (OCT.88).

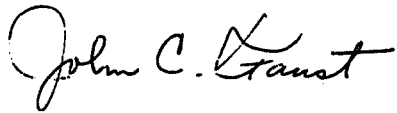
AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

20020116 187

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-172 has been reviewed and is approved for publication.

APPROVED:



JOHN C. FAUST
Project Engineer

FOR THE DIRECTOR:



WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

COMPOSABILITY FOR SECURE SYSTEMS

Duane Olawsky, Charles Payne,
Todd Fine, Tom Sundquist,
And David Apostol

Contractor: Secure Computing Corporation
Contract Number: F30602-96-C-0344
Effective Date of Contract: 17 October 1996
Contract Expiration Date: 20 November 1998
Short Title of Work: Composability for Secure Systems
Period of Work Covered: Oct 96 - Nov 98

Principal Investigator: Duane Olawsky
Phone: (612) 628-2846
AFRL Project Engineer: John C. Faust
Phone: (315) 330-4544

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION
UNLIMITED.

This research was supported by the Defense Advanced Research
Projects Agency of the Department of Defense and was monitored
by John C. Faust, AFRL/IFGB, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE AUGUST 2001		3. REPORT TYPE AND DATES COVERED Final Oct 96 - Nov 98
4. TITLE AND SUBTITLE COMPOSABILITY FOR SECURE SYSTEMS			5. FUNDING NUMBERS C - F30602-96-C-0344 PE - 62301E/33401G PR - D849 TA - 00 WU - 01	
6. AUTHOR(S) Duane Olawsky, Charles Payne, Todd Fine, Tom Sundquist, and David Apostol				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Secure Computing Corporation 2675 Long Lake Road Roseville MN 55113			8. PERFORMING ORGANIZATION REPORT NUMBER Part Number 00-0-321A-Rev B	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) DARPA National Security Agency AFRL/IFGB 3701 N. Fairfax Dr. NSA/R2SPO 525 Brooks Road Arlington VA 22203-1714 9800 Savage Road Rome NY 13441-4505 Fort Mead MD 20755-6000			10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2001-172	
11. SUPPLEMENTARY NOTES Air Force Research Laboratory Project Engineer: John C. Faust/IFGB/(315) 330-2665				
12a. DISTRIBUTION AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This document is the final technical report for the Composability for Secure Systems (CSS) program whose objective was to develop and demonstrate a composable methodology for building highly-assured, fault tolerant distributed systems and networks, and to design an automated development environment to support the methodology. The CCS program studied composition and refinement as unifying concepts that support analysis of functional correctness, fault tolerance and security form a single specification. A mathematical framework was developed for specifying and analyzing systems. This report provides an overview of all the technical conducted in pursuit of these goals. It described significant accomplishments and obstacles encountered during these tasks and lessons learned while carrying out the program. Finally, it provides suggestions for future effort which build upon the success of the program or addresses deficiencies.				
14. SUBJECT TERMS Composition, Design Refinement, Formal Methods, Computer Security, Fault Tolerance, Restrictiveness, Non Interference, PVS, Formal Specification Tools, Formal Analysis Tools			15. NUMBER OF PAGES 64	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Contents

1	Scope	1
1.1	Identification	1
1.2	Document Overview	1
2	Program Summary	2
2.1	Objectives and Approach	2
2.2	Significant Results and Accomplishments	2
2.3	Technical Documentation	3
2.4	Dependent Programs	4
3	CSS Framework	5
3.1	Goals	5
3.2	Approach	6
3.3	Accomplishments	8
3.4	Lessons	11
3.5	Future Work	13
4	Top Level Specification	14
4.1	Goals	14
4.2	Approach	14
4.3	Accomplishments	14
4.4	Lessons	16
4.5	Future Work	16
5	Design Refinement	17
5.1	Goals	17
5.2	Approach	17
5.3	Accomplishments	18
5.4	Lessons	19
5.5	Future Work	20
6	Fault Tolerance	21
6.1	Goals	21
6.2	Approach	21
6.3	Accomplishments	28
6.4	Lessons	28
6.5	Future Work	29
7	Policy Composition	30
7.1	Goals	30
7.2	Approach	31
7.3	Accomplishments	34
7.4	Lessons	36
7.5	Future Work	36
8	Tool Support	37
8.1	Goals	37

8.2	Approach	37
8.3	Accomplishments	43
8.4	Lessons	46
8.5	Future Work	48
<hr/>		
9	Program Conclusions	50
<hr/>		
A	Bibliography	51

List of Figures

1	The Fault-Free Model	23
2	The Fault Tolerant Model	23
3	Fault Tolerant TLS Architecture	27
4	System-level diagram.	39

Section **1** **Scope**

1.1 Identification

This document is the final technical report for the Composability for Secure Systems program, contract F30602-96-C-0344. It provides an overview of all of the technical efforts on the program. It describes significant accomplishments and obstacles encountered during these efforts and lessons learned while carrying out the program. Finally, it provides suggestions for future efforts which build upon the successes of the program or address deficiencies.

1.2 Document Overview

The report is structured as follows:

- Section 1, **Scope**, defines the scope and gives this overview of the document,
- Section 2, **Program Summary**, provides a high-level summary of the objectives, approach, and results of the CSS program,
- Section 3, **CSS Framework**, describes the CSS composition and refinement framework,
- Section 4, **Top Level Specification**, describes the Top-Level Specification (TLS) which models network-transparent IPC,
- Section 5, **Design Refinement**, summarizes the refinement of the network server component of the TLS into an *x*-Kernel network stack,
- Section 6, **Fault Tolerance**, describes the application of the framework to the specification and analysis of fault tolerance properties,
- Section 7, **Policy Composition**, describes the modeling of restrictiveness in the framework to provide for composable non-interference analysis,
- Section 8, **Tool Support**, describes prototype tools to support the application of the framework,
- Section 9, **Program Conclusions**, summarizes the main conclusions of the CSS program,
- Appendix A, **Bibliography**, gives citations for each referenced document.

Program Summary

2.1 Objectives and Approach

The objective of the Composability for Secure Systems (CSS) program was to develop and demonstrate a composable methodology for building highly-assured, secure, fault-tolerant distributed systems and networks, and design an automated development environment to support the methodology.

Many kinds of system properties may be formally analyzed during a system design effort including functional correctness, fault tolerance and security. Substantial work has been done to establish effective methods for performing each of these kinds of analysis. However, it is typically the case that each method requires a particular style or language for the underlying formal description of the system. This is unfortunate for two reasons. First, there can be significant expense in producing multiple specifications of a system. Second, it introduces the question of whether all the specifications are consistent, and it makes it harder to determine whether an implementation of the system is correct since it must be compared to multiple formal descriptions.

To address these issues the CSS program has studied composition and refinement as unifying concepts that support analysis of functional correctness, fault tolerance and security (non-interference) from a single specification. A mathematical framework has been developed for specifying and analyzing systems. The program has demonstrated the application of this framework to the analysis of functional correctness, fault tolerance and security.

2.2 Significant Results and Accomplishments

The following list highlights the major results and accomplishments of the program:

- Developed a framework in PVS that supports composition and refinement reasoning including fairness properties
- Used the framework to perform a proof-of-concept analysis of fault tolerance
- Formalized the restrictiveness information flow policy within the framework and proved that this formalization is composable
- Explored the development of tools to support application of the framework including a specification browser and extensions to the PVS prover to make it easier to perform proofs within the framework
- Specified an α -Kernel protocol stack as a composition of components within the framework and used the framework to show that this stack is a refinement of an abstract network server specification

2.3 Technical Documentation

There are four primary types of output from the CSS program: technical reports, a short research paper to be presented at the Third IEEE High Assurance Systems Engineering Symposium in November, PVS libraries, and software for the prototype tools. The technical reports, PVS libraries and software are available from the CSS Web Page (<http://www.securecomputing.com/css/>). The libraries are in the form of PVS dump files. The software consists of Emacs LISP code implementing the Specification Browser tool and PVS strategies implementing the Analyst's Assistant tool. The libraries and software are described in the corresponding technical reports. This section presents brief descriptions of all technical reports written for the program. It is divided into sections for contract deliverables (CDRLs) and a short research paper.

2.3.1 CDRL Document Summary

This section describes all technical reports provided as contract deliverables (CDRLs). The missing CDRLs in this list are for the program management documents.

Top-Level Specification (TLS), CDRL A004 [15]

In this report we give a high level description of a network-transparent, distributed interprocess communication manager. This report provides a baseline specification to be used in other CSS tasks, and serves as an example starting point for the CSS design methodology.

Security Policy Composability Results (SPCR), CDRL A005 [21]

We describe tools and a methodology for analyzing a noninterference property called *restrictiveness* in the CSS composition and refinement framework. These tools take the form of extensions to the composition and refinement framework [14] that are suited for performing security analyses of systems at the component level.

Refined Design Report (RDR), CDRL A006 [14]

This report serves several purposes relating to the use of our composition and refinement framework. The bulk of the report exhibits a working example of refinement; the top level specification of a network server (described in the TLS [15]) is here refined into a detailed description of network protocols using the α -Kernel methodology [5]. The document also discusses the PVS theories comprising the framework and relates lessons learned while designing, implementing, and testing the framework.

Tools Report, CDRL A007 [22]

This report contains a description of the requirements, design and code for prototype tools developed on the program. Two types of tools—a browser for viewing and editing specifications, and PVS strategies to help in the formal analysis of specifications—are covered.

Modified Top-Level Specification (MTLS), CDRL A008 [20]

This report presents a modified version of the Top-Level Specification (TLS) that includes fault tolerance properties.

Fault Tolerance Analysis (FTAR), CDRL A009 [18]

This report presents the results of a proof-of-concept effort to reason about fault tolerance properties using the refinement framework. An architecture for fault tolerance is specified and analyzed.

Final Report, CDRL A010 [19]

This report is the final report for the CSS program and provides an overview of all of the technical efforts on the program. It describes significant accomplishments and obstacles encountered during these efforts and lessons learned while carrying out the program. Finally, it provides suggestions for future efforts which build upon the successes of the program or address deficiencies.

2.3.2 Research Paper

Using Composition to Design Secure, Fault-Tolerant Systems

This paper summarizes the program's approach and results.

2.4 Dependent Programs

Two other research programs at SCC have benefited from the CSS program.

- **Assurance in the Fluke Microkernel (AFM) (June 1997 – present)**

The CSS Framework is currently being used in two ways on the AFM program [17, 16]. A simplified process manager and file server are being specified and analyzed with the goal of further demonstrating and exploring the use of the framework. At least two levels of abstraction will be used: an abstract requirements level and a more detailed “implementation level.” The desired properties of secure process creation will be demonstrated at the requirements level. Lower levels will be shown to be refinements of this level, therefore inheriting the desired properties. The AFM program is also using the CSS Framework to perform a general analysis of issues involved in maintaining consistency between an object manager that enforces a security policy and a security server that supplies security decisions based on that policy, paying special attention to the case where the policy can change and the object manager caches policy decisions.

- **Hypervisors for Security and Robustness (October 1996 – March 1998)**

This program used the CSS composition and refinement framework to support the specification and analysis of the composition of a kernel with *security hypervisors* that monitor kernel requests to provide security [23, 24].

Section 3

CSS Framework

3.1 Goals

The primary goal of this portion of the program was to develop a formal framework for reasoning about composite systems and performing formal design refinement. This provided an underlying framework that served as a basis for the remainder of the work on the program. Specific goals in developing the framework were to make it small, easy to use and not too hard to verify while still providing the essential reasoning power.

Composition is a technique for specifying a large system by combining the specifications of smaller, simpler pieces—the system components. This technique provides advantages that are similar to those obtained from modular software design. The smaller, simpler pieces are typically easier to write and maintain. This technique also allows for reuse of specifications for individual components. Just as a well-designed software module can be reused in a variety of new contexts, a component specification can be combined with other component specifications in novel ways to obtain new system specifications. However, the benefits do not stop there. We typically wish to analyze a system specification to show that it satisfies certain desired properties. Composition allows us to decompose this analysis into the analysis of the components. Rather than analyzing the entire composite system, we focus on a single component at a time, showing that it satisfies some more localized property. We then show that the localized properties of the components together imply that the global desired property is satisfied by the system as a whole. Since the local analyses depend upon only a single component, they are not only simpler but also reusable. They need not be redone when the component is used in a new context. In addition to the composition of *pure* analyses described above, we can also compose *conditional* analyses. In this case, we show that a component satisfies a localized property under given assumptions about the environment. When we compose this component with others we show that the other components justify the environment assumptions. One can then conclude that the system as a whole satisfies the localized property.

Refinement supports reasoning about a system specified at multiple levels of abstraction. Refinement analysis makes it possible to show that properties demonstrated for an abstract specification are preserved in a less abstract refinement of that specification that reflects a more detailed design. It is usually easier to prove desired system properties for an abstract specification than for a more detailed one. On the other hand, it is easier to relate a detailed specification to its implementation since it includes more design details. The goal, of course, is to know that the implementation satisfies the desired properties. Refinement analysis allows us to conclude this by proving that an abstract specification satisfies the properties, then arguing that a refined specification is consistent with the implemented system and finally, comparing the two specifications to show that the refined one is consistent with the abstract one. Thus, we only need analyze the property at the abstract level where this analysis is easiest. As with composition, refinement can be *pure* (no assumptions about the environment are needed) or *conditional* (the implementation argument depends on assumptions about the environment that are later justified by other components).

Composition can be thought of as bottom-up analysis; properties shown to hold for components are true of the composite system. Refinement is top-down analysis; properties demonstrated

at the abstract level are shown to be true of more detailed levels as well.

In this section we describe the CSS Framework, a mathematical framework that supports the following:

- specifying system components (Section 3.2.1),
- composing components (Section 3.2.2), and
- performing both composition and refinement reasoning (Section 3.3.1).

Comparisons to prior work are included in Section 3.3.2. The framework is formalized in the PVS specification language [9], and all results have been proven in PVS.¹

3.2 Approach

3.2.1 Component Definitions

We begin by outlining the information that comprises a component specification in the CSS framework. Each component specification c is a record defining the actions that the component is willing to perform and placing constraints on the actions its environment is able to perform. For each component, the set $cags(c)$ (short for component agents) denotes a set of agents that mediate actions of that component. The set $guar(c)$ (guarantee) denotes the transitions that the component can perform. A transition is modeled by a triple (s_1, s_2, a) indicating a change of state from s_1 to s_2 mediated by agent a . Since $guar(c)$ contains transitions performed by c , the agent of each $guar$ transition must be in $cags(c)$.

The set of transitions $hidd(c)$ (hidden) denotes the set of external (environment) transitions allowed by the component. The agent in a $hidd$ transition must not be in $cags(c)$. The name $hidd$ is deceptive. It suggests data or transitions that are private and cannot be observed by other components. However, $hidd$ really denotes a set of transitions allowed in the environment. Visibility plays no essential role. The term $hidd$ dates from early versions of the framework and has been kept primarily for historical reasons and because of the inertia in a large formal system.

Examples of constraints that $hidd$ might place on the environment include

- No environment agent may alter the internal variables of c .
- Only agent a may alter the interface that c presents to a .

These constraints are included in the component specification for c because they are critical for the correct operation of c . The specification of $hidd$ makes composition possible. Section 3.2.2 below describes how $hidd$ is applied during composition.

For convenience, we define $steps(c) = guar(c) \cup hidd(c)$. The set of initial states allowed by c is denoted by $init(c)$. Fairness requirements are represented by two fields, $wfar(c)$ and $sfar(c)$, corresponding to weak and strong fairness conditions. Fairness requirements provide a way to abstractly specify system liveness and scheduling properties. Each fairness condition is represented by a set of transitions; each of $wfar(c)$ and $sfar(c)$ contain a set of fairness

¹The framework will be available as a PVS dump file on the CSS Web Page <http://www.securecomputing.com/css/>. The framework is described in detail toward the end of the CSS Refined Design Report [14].

conditions (i.e., a set of sets of transitions). An execution history (i.e., trace) t for the system satisfies a weak fairness condition $F \in wfar(c)$ if an infinite number of F -transitions occur in t or there are an infinite number of states in t in which F -transitions are not enabled (i.e., there is no transition in F that starts in the given state). Similarly, an execution history t satisfies a strong fairness condition $F \in sfar(c)$ if an infinite number of F -transitions occur in t or there is some point in t after which F -transitions are never enabled.²

These six fields define the set ($mprop$) of execution histories allowed by the component:

$$mprop(c) = init(c) \cap (\Box steps(c)) \cap fair_prop(c),$$

where $fair_prop(c)$ denotes the histories that satisfy all the weak and strong fairness conditions of c , and \Box is the temporal operator “always.” This formula denotes the set of all execution histories that start in a state in $init(c)$, that contain only transitions in $steps(c)$ and that satisfy $fair_prop(c)$. This set of histories is called the *property* of c .

The two remaining fields of a component definition are *view* and *rely*. The former describes the portion of the state that is “visible” to the component. This does not directly enter into the property definition and is used only to validate that the other fields of the component are defined solely in terms of a specific portion of the system state information. The *rely* field is used only for conditional analysis of components. This is discussed in Section 3.3.1.

3.2.2 Composition

The basic idea of composition is

- the composed components start in a common state that is acceptable to all of them,
- the components take turns performing transitions that are allowed by all the components³, and
- the fairness conditions of all the components are satisfied.

We define the expression $compose(S)$ to denote the composition of the components in the set S . A composite system is itself a component. Let $d = compose(S)$. Then

$$\begin{aligned} init(d) &= \bigcap_{c \in S} init(c) \\ wfar(d) &= \bigcup_{c \in S} wfar(c) \\ sfar(d) &= \bigcup_{c \in S} sfar(c) \\ cags(d) &= \bigcup_{c \in S} cags(c). \end{aligned}$$

We want $steps(d) = \bigcap_{c \in S} steps(c)$ (i.e., d allows only steps that are allowed by all components). Using $cags(d)$, we break this intersection into the desired *guar* and *hidd* for d as follows. We take

$$hidd(d) = \bigcap_{c \in S} hidd(c).$$

²See [14] for more information on $wfar$ and $sfar$ and [6] for a general discussion of specifying and reasoning about fairness properties.

³We consider only interleaving representations of concurrency.

This set includes exactly the transitions in $steps(d)$ that do not have an agent in $cags(d)$. We take

$$guar(d) = steps(d) - hidd(d),$$

(i.e., the set of all transitions in $steps(d)$ having an agent in $cags(d)$). Equivalently, $guar(d)$ is the set of all transitions that are in $guar(c)$ for some $c \in S$ and in $steps(c)$ for all $c \in S$ (i.e., those transitions that can be performed by at least one component and are allowed by all components). Note that we allow components to have overlapping $cags$. However, if for every pair of components in S the $cags$ sets are disjoint, then $guar(d)$ is the set of all transitions that are in $guar(c_1)$ for some $c_1 \in S$ and in $hidd(c_2)$ for all $c_2 \in S, c_2 \neq c_1$. This means that the $hidd$ of each component is automatically applied to the $guar$ of each of its peer components during the composition of the components.

This automatic application of the $hidd$ values is important from a standpoint of reuse of specifications. It removes the need to modify the specification of a component whenever it is to be composed with a new component. It also allows the definition of a component to focus entirely on its own state. This approach does however give $hidd$ great power. There is no way in this framework for the $hidd$ of a component to be violated by one of its peers in a composite system. This is why we stress that the $hidd$ is to be considered part of the specification and any valid implementation of the component must obey the requirements expressed in $hidd$. It is every bit as important as the $guar$ in achieving a faithful implementation of the component. In analyzing an implementation for faithfulness to the $hidd$, it may be necessary to consider other pieces of software. For example, the $hidd$ of a component might say that certain local variables are not altered during environment transitions. An implementation might achieve this by placing these variables in a particular region of memory and relying on the kernel to

- protect this memory from other processes (i.e., to maintain address-space separation), and
- not alter the memory itself.

Of course, the component itself must not do anything that would subvert the $hidd$ such as asking the kernel to share the private region with other processes.

3.3 Accomplishments

3.3.1 Composition and Refinement Reasoning

In this section we describe several theorems for reasoning about components and their compositions and refinements. In addition to these specific theorems, the framework provides a wide variety of theorems for reasoning about general system properties including fairness properties. These theorems can make analysis of component properties easier. They can even make it easier to perform general requirements analyses independent of any component specifications since they incorporate the mathematical inductions that are often necessary in such analyses. For example, an analyst can apply a single theorem to reduce the proof of a state invariant to a proof that the initial state satisfies the invariant and every allowed transition maintains the invariant.

Since the result of composition is a component, any place we refer to a component a composition of components can be substituted.

Once a component is specified, one typically wants to prove that it satisfies certain desired properties. As noted above, a property is a set of execution histories. A component c is said to

satisfy a property P if $mprop(c) \subseteq P$. The following theorem clarifies the relationship between *compose* and *mprop*:⁴

Theorem 1 (*mprop* Intersection) For any set S of components,

$$mprop(compose(S)) = \bigcap_{c \in S} mprop(c)$$

It is a trivial consequence of this theorem that if c satisfies P , then so does every composite system containing c as a component. This supports reuse of analysis and the decomposition of a satisfaction proof into smaller, component-local satisfaction proofs.

Frequently, a software component is designed so that it satisfies some desired property as long as certain assumptions hold true for its environment (e.g., the processes with which it communicates follow an agreed upon protocol). In this case, we say the component *conditionally satisfies* the property. To show that it satisfies the property unconditionally, we must show that the environment justifies the assumptions. In the CSS Framework, the necessary assumptions are stored with the component in the *rely* field which contains a set of environment transitions. Unlike *hidd*, this is not a constraint upon an implementation of the component. It is merely an assumption that supports conditional satisfaction analysis. A component c *conditionally satisfies* property P if P contains every execution history in $mprop(c)$ such that every transition is either in $rely(c)$ or has an agent in $cags(c)$. The following theorem makes it possible to convert a conditional satisfaction result for some component to an unconditional one by composing the component with an environment that justifies the assumptions:⁵

Theorem 2 (Composite Satisfies) For every set of components S , component $c \in S$ and property P , if

1. c conditionally satisfies P , and
2. every transition $r \in steps(compose(S))$ is either in $rely(c)$ or has an agent in $cags(c)$,

then $compose(S)$ satisfies P (unconditionally).

Note that the first hypothesis depends only upon c and P . Therefore, its analysis can be reused when c is composed into a new environment. The second hypothesis is sensitive to the removal or modification of components in S , but not to their addition. It is thus reusable in certain circumstances. Also, this hypothesis does not require an analyst to consider all components in S . Often, one or two components in S supply all the restrictions on r necessary to satisfy the hypothesis.

There is an alternative to conditional satisfaction analysis that is also supported by the framework. In this approach, one proves that a component satisfies a *conditional property* of the form $P \Rightarrow Q$ where P and Q are both properties. One must show that every execution history allowed by the component (ignoring any assumptions added by *rely*) either is not in P or is in Q . If the goal is to show that the entire system satisfies Q , then the component must be composed with one or more other components that restrict the allowed execution histories to those in P . This approach is actually stronger than conditional satisfaction since P can contain initial state restrictions and fairness conditions in addition to transition restrictions as can be specified in *rely*.

The framework also supports refinement analysis. A component c is a *refinement* of (or *implements*) another component d if $mprop(c) \subseteq mprop(d)$. This is a useful concept since property

⁴ The name of this theorem in the PVS framework is *mprop_conj.thm*.

⁵ This is theorem *compose_satisfies* in the PVS.

satisfaction is preserved under refinement. Thus properties can be proven at an abstract level of specification and a refinement analysis can then be performed to show the properties remain true for a lower level specification. Since $mprop(c)$ and $mprop(d)$ are typically infinite sets of execution histories, which are themselves infinite sequences, an arbitrary refinement proof can be difficult. The following theorem reduces the proof to sufficient conditions that are much easier.⁶

Theorem 3 (*mprop* Implementation) For any two components c and d , c implements d if

1. $init(c) \subseteq init(d)$
2. $steps(c) \subseteq steps(d)$
3. $mprop(c) \subseteq fair_prop(d)$

Note that only the third hypothesis requires reasoning about infinite sequences. The first two hypotheses deal with states and transitions. Numerous framework theorems regarding fairness properties support the proof of the third hypothesis, but there is insufficient space to describe these here. (See [14].)

When c and d are both composites, the proof that c implements d can be decomposed into a collection of smaller proofs showing that for each component e of composite d , there is some set S_e of the components comprising c such that $compose(S_e)$ implements e . The subproofs will usually be easier than the large one. Furthermore, each subproof that $compose(S_e)$ implements e is likely to be reusable in another refinement analysis where e is implemented by the same low-level components S_e but is composed with a different set of components. The CSS Framework contains two theorems supporting such decompositions. The first, *pure_decomp.thm*, supports *pure decompositions* in which the subproof for each e is entirely independent of the components not in S_e . The analyst only need perform each of the subproofs.

The second theorem, *decomp.thm_reduced*, supports *impure decompositions* where at least one of the subproofs requires assumptions about components outside the subproof. An impure decomposition is sometimes necessary if the granularity of transitions changes from one specification level to the next. As with conditional satisfaction, we use *rely* to store the assumptions about the other components. Assume the proof that c implements d is impurely decomposed into n subproofs that c_i implements d_i , $i = 1, \dots, n$. The following must be demonstrated for each subproof i .

1. The refinement is valid assuming $rely(c_i)$ is never violated (we call this *conditional implementation*),
2. $hidd(c_i) \subseteq hidd(d_i)$, and
3. every transition $r \in steps(compose(\{d_1, \dots, d_n\}))$ is either in $rely(c_i)$ or has an agent in $cags(c_i)$ (since the d_i are more abstract than the c_i , this will typically be easier than justifying the assumptions in terms of the c_i).

The first two hypotheses depend only upon the components involved in subproof i and are therefore reusable. The third hypothesis can be invalidated by the removal or modification of components composed into d , but not by the addition of new components. It is thus partially reusable. As with conditional satisfaction, the analyst need not consider all the d_i in this hypothesis, only those d_i that provide the necessary information.

⁶This is theorem *mprop.impl.thm* in the PVS.

3.3.2 Related Work

The CSS Framework is descended from an earlier version developed under the Distributed Trusted Operating Systems (DTOS) program [12, 4]. The DTOS framework dealt with composition only and had no support for refinement reasoning. It also had a more restrictive definition of a component's property in which *rely* was used in place of *hidd*. In addition to asserting that the environment assumptions must be satisfied, this had the side effect of introducing an $O(n^2)$ proof obligation (where n is the number of components) in the DTOS version of the *mprop* Intersection theorem. An analyst had to show that for every pair of components (c, d) being composed, $\text{guar}(c) \cap \text{hidd}(d) \subseteq \text{rely}(d)$.

The DTOS and CSS Frameworks are both heavily influenced by the composition work of Abadi and Lamport [2] which is couched in the Temporal Logic of Actions (TLA) [6] and by Shankar's composition framework [25]. TLA specifications (called *formulas*) are similar to CSS components except that

- they do not have explicit agents,
- the *hidd* is effectively an equivalence relation on states, and
- formulas may include existential quantifiers to hide internal state.

We added agents because the performer of an action is important in certain types of security analysis, one of our primary applications. Having agents also gives us the flexibility of distinguishing between environment agents in our *hidd*. This especially makes sense when specifying a kernel which can determine the identity of its clients and keep them separate. We explored the introduction of quantifiers into our framework but decided against them. They forced a great deal of complexity into the framework in terms of both verifying the framework and using it. Having quantifiers would have introduced an $O(n^2)$ proof obligation into the *mprop* Intersection theorem to show that no two components quantify over the same variable. Although quantification has advantages from a philosophical standpoint, its practical value is more suspect. In TLA the first step in a refinement proof is typically to remove the quantifiers by applying a *refinement mapping*. This step puts the proof at what is the starting point for the proof in our framework. If no refinement mapping can be found, then, although the refinement may still be correct, the proof is likely to be very difficult. So, our framework makes it easier to do the refinement proofs that are likely to be feasible at the expense of not supporting refinement proofs that are likely to be very difficult. Finally, in TLA, composition does not automatically apply the *hidd* as is done in our framework. Before applying the TLA equivalent of the *mprop* Intersection theorem the TLA formulas must be modified "by hand". Then $O(n^2)$ proof obligations must be dispensed to show that the hand modifications are correct.

3.4 Lessons

The most important lessons from the development of the CSS Framework are reflected in its current structure and in technical details such as not supporting quantification in component specifications.

In our first approach to the framework we attempted to follow the Abadi-Lamport work [2] rather closely, translating their concepts, results and proofs into PVS. These very general results would then be applied to the specific style of component specifications used in our framework. We expected this approach to be easier than trying to prove similar results on our own. Through our efforts to achieve this, we gained a much better understanding of the

Abadi-Lamport results. Eventually, we encountered some significant difficulties in following this approach, but the understanding we gained through our attempts allowed us to overcome these difficulties by approaching the problem from a different direction. The difficulties we encountered stemmed from

- the central role of *invariance under stuttering* in the Abadi-Lamport work and the difficulty of dealing with this concept in PVS,
- the complexity of including existential quantification in the framework, and
- the large size of the PVS LISP image.

We describe each of these difficulties briefly along with how we avoided them in our second approach and how this contributed to the framework goals of ease of use, ease of verification and sufficient reasoning power.

One of the assumptions in the Abadi-Lamport work is that the properties considered are *invariant under stuttering*. This means that if a behavior t_1 satisfies some property, then so does every behavior t_2 obtainable from t_1 by adding and deleting an arbitrary (possibly infinite) number of stuttering steps (i.e., steps in which no state change occurs). Although this concept is fairly intuitive, it is somewhat complicated to work with in PVS. A good deal of mathematical machinery would be needed in the framework to effectively exploit this assumption in the framework proofs. Although it is frequently trivial to *informally* compare two infinite sequences to see whether they are equivalent up to stuttering, doing this *formally* is usually a good deal more challenging, especially when there can be an infinite number of places where stuttering steps have been added or removed. However, the components in our framework trivially satisfy invariance under stuttering since the *guar* and *hidd* are required to contain all stuttering steps. When proving theorems about arbitrary composite systems it is much easier to work with the *guar* and *hidd* which are sets of transitions than to apply the more abstract notion of invariance under stuttering which deals with sets of infinite sequences. Thus, it is much easier to take components (and the properties they define) as the starting point than to use the more general concept of properties invariant under stuttering.

While attempting to follow the Abadi-Lamport approach we did search for a simplification of invariance under stuttering that would be easier to work with in PVS. For example, we attempted to use a definition that allowed only a finite number of stuttering steps to be added or deleted. However, it was eventually discovered that the simpler definition is insufficient when dealing with existentially quantified properties (to be discussed shortly) since the quantification can introduce an infinite number of stuttering steps. Furthermore, the simplified definitions were still not very easy to use in performing the verification of the framework in PVS. In the end, we abandoned the concept of invariance under stuttering entirely, relying instead on the constraints on *guar* and *hidd* in the framework. This decision significantly reduced the size and complexity of the framework, and this contributed to the goals ease of use and ease of framework verification. This decision had little if any effect on the formal power of the framework since the intent had always been to provide a component-based framework to the analysts. Concepts such as invariance under stuttering were included only to support the incorporation of the Abadi-Lamport proofs as general results of which the component-based theorems would be corollaries.

A second difficulty encountered in our original approach was the complexity of quantification in specifications. Quantification supports data hiding in a specification. For example, it allows one to specify the external behavior of a queue (i.e., first in, first out) in terms of a convenient internal representation (e.g., a sequence) without making that representation externally visible. By not including quantification, we lose the ability to hide the internal representation.

Of course, we can still prove within the framework that designs using two different representations are equivalent by showing how the representations relate to each other. Since this is essentially what happens in a framework that supports quantification when a refinement mapping is applied to quantified specifications during analysis, we are really just requiring that the refinement mapping be specified when the specifications are placed in a common state space for analysis rather than during the refinement proof itself. This may well be a feature of the framework since it makes the refinement mapping (i.e., the relationship between data at different specification levels) explicit in the specifications rather than hiding it in the refinement proof.

It was also realized that quantification would significantly increase the burden on users of the framework. When reasoning about composite systems, analysts would be required to show that no two of the components quantified the same variable. This would be a pairwise, $O(n^2)$ proof obligation. As with invariance under stuttering, the decision to avoid quantification improved ease of use and verification, and it reduced the size of the framework. Any practical reduction in reasoning power is minimal since it is usually quite difficult to perform a refinement proof without using a refinement mapping. Not having quantifiers makes it impossible to do certain refinement proofs that are probably impractical to perform anyway. At the same time it simplifies the proofs that are practical.

We have already mentioned that the decisions to focus on components rather than stuttering-invariant properties and to not support quantification resulted in a smaller framework. Aside from concerns of ease of use and learning curve, this had some very practical hardware resource implications. In the original approach we noted that the PVS Lisp image was getting quite large (e.g., 50-60 Meg for just the framework). We realized that this might have an impact on usability of the framework both in terms of the hardware demands and the slower speed of PVS when the image gets large. We were able to significantly reduce the image size in going to the new approach.

3.5 Future Work

As noted in Section 2.4, the Assurance in the Fluke Microkernel program is currently using the CSS framework to analyze secure process creation and to study the support of dynamic security policies in a distributed environment.

One goal of the CSS program was to demonstrate that composition and refinement could be used for a variety of analyses necessary in designing systems, especially secure systems. It would be interesting to continue this demonstration by considering other types of analyses. For example, the framework could be used to study analysis of real-time properties (see [1]). This not only has broad relevance outside security, but could also be applied to the analysis of security mechanism such as real-time audit. Another area that might be fruitful is the analysis of cryptographic protocols, especially in an environment where the protocol is implemented by cooperating (and potentially reusable) components.

Finally, we anticipate that a variety of evolutionary improvements could be made to the framework itself. It could be fine-tuned for ease of use, clarity of organization, quality and quantity of documentation, small size, etc.

Top Level Specification

4.1 Goals

The Top-Level Specification (TLS) effort was intended to serve several purposes, the foremost of which was to provide a starting point for the design refinement work and the portion of the fault tolerance effort in which an existing specification (the TLS) is modified to be fault tolerant. The TLS also serves as an example of how to specify system components within the CSS Framework. In doing this we took the position (shared by earlier programs such as DTOS [13]) that a component specification has multiple audiences and ought to be readable by all of them. Thus, it should contain not only formal notation but also other representations of the specification such as English text and processing tables (e.g., case coverage tables). We furthermore took the stance that a formal system model ought to be refined in parallel with the system design effort and that throughout this refinement process it is important to keep the formal and informal descriptions consistent.

4.2 Approach

The chosen functionality to be specified in the TLS was network-transparent, distributed, interprocess communication (IPC). To better explore the refinement of a design from high-level requirements to a detailed design we included two levels of abstraction within the TLS itself (the network server from the second level is further refined in the Refined Design Report). The high level specification consists of a single component called the Box Manager. The purpose of this specification is to describe the high-level requirements that characterize IPC. Communication is controlled by capabilities which may be transmitted in messages. The Box Manager maintains these capabilities and manages communication *boxes* that serve as containers for messages in transit. The refined description of the system contains components for kernels, network servers, and a network. This level spells out the details of a particular refinement of the Box Manager into a networked system with multiple nodes (kernels), network servers and a network. The network server components act as forwarding agents that forward node-local messages to network servers on remote nodes for delivery to the appropriate remote client and that forward messages received from the network to the appropriate local client.

To address the goal of close correspondence between formal in informal representations, we organized each of the four component specifications in the TLS as a series of pairs of text and formal specification (PVS) elements. This makes it easier to visually inspect the descriptions for consistency. We also included case-based processing tables. (See Section 8 for information on the Specification Browser which further encourages consistency between the descriptions.)

4.3 Accomplishments

The TLS was the first attempt at specifying a system using the CSS Framework at two levels of abstraction. Having two levels proved to be very useful. First, it allows the critical properties of the system to be proved at a more abstract level where the proofs will typically be much

easier. Meanwhile, the more detailed level reflects design decisions that will eventually be followed in an implemented system. This level has a stronger resemblance to the implementation. Combining these two levels and showing that one is a refinement of the other thus provides important guidance and assurance to the design effort. No significant problems were encountered in specifying the system at multiple levels although we did not have sufficient time to perform a PVS proof that the kernel-network level is a refinement of the Box Manager.

The TLS was also the first attempt at defining and composing components without the use of state and agent translators which were used heavily in applications of the DTOS Framework. The framework requires that all components to be composed are defined on a common “universal” state. However, when defining a single component it would be undesirable to have to anticipate the state information of all the other components with which the current component might someday be composed so that that state information can be included. This would make it much harder to reuse component specifications in new ways and would thus reduce one of the main benefits of the framework. So, the approach in the DTOS framework was to define each component on an separate state type, define a global state that contains all the state information, and then define translator functions that map each local state to the global state. It was noted on DTOS that this was fairly clumsy. In addition to the work of defining the translators, significant portions of the proofs about the system dealt with “applying” the state translations. Since the translations themselves were usually trivial project functions, there was no value obtained for this effort. Furthermore, it was realized that if we ever used non-trivial translators, the translators themselves could obscure the true nature of what was being proven about the system.

So we developed an alternative approach that has been used throughout the CSS work—and is also being used in the Assurance in the Fluke Microkernel program (see Section 2.4)—that does not require translators.⁷ This approach has worked quite well. As in the translator approach, for each type of component we define a local state type, ignoring possible overlaps with other components. Shared data types and global “configuration” information are defined in a low level PVS theory called *config* which is imported by all other state models. The local state definitions are combined in the theory *common.state* with a separate field for each local state. Each component is defined directly on the common state but all of its accesses are to its local portion of that state via the appropriate field accessor function. Thus, the component really only depends upon its local state. As components are added or removed, the changes to global state are isolated to the two theories *config* and *common.state*. Since each component is defined with respect to one field of the common state, the addition or removal of fields has no effect on the other components nor their proofs. This provides the desired reusability of specifications without the use of translators.

In the fault tolerance work we considered using the Box Manager as an abstract specification of communication between the fault tolerance components. However, the fault tolerance communication requires first-in first-out (FIFO) communication that this is not specified in the Box Manager. Thus, the specification of IPC in the Box Manager may be too general for some potentially profitable uses. FIFO was not needed for the critical properties that were anticipated when writing the Box Manager, and omitting this requirement resulted in a simpler specification. This is likely to be a perpetual question in writing specifications: what requirements ought to be specified to give the specification broad application without making it too large, complex or cluttered.⁸

⁷ In fact, the concept of translators has been removed entirely from the framework.

⁸ On the Assurance in the Fluke Microkernel program we are currently exploring an approach that could be used to alleviate this problem. New requirements can often be added by composing in a new system component at the requirements level that further constrains the system to obey the new requirement. Thus, a FIFO component could assert that the communication mediated by the Box Manager must be FIFO.

4.4 Lessons

The lessons from this task relate closely to the achievements described in the previous section:

- State and agent translators are not necessary.
- Specifying a system at multiple levels of abstraction has significant advantages.
- It is not always easy to reuse a specification because the needs of its new use might not match those of its earlier use.

The first lesson is important for the usability of the CSS Framework. Given prior experience in the field of formal methods, the second and third lessons hardly seem surprising.

4.5 Future Work

One thing that we had hoped to do but for which we had insufficient time was to perform a PVS proof that the kernel-network level is a refinement of the Box Manager. We expected to learn some things about refinement arguments and about high-level specifications by doing this. In particular, this refinement step moves from a high level in which clients can directly refer to boxes to a low level in which they reference them indirectly through names in a name space that the kernel maintains for each client. This is likely to introduce complexities into the refinement mapping. Furthermore, we expect this refinement from direct to indirect references to be a common theme in refinement arguments since it is much easier to analyze critical properties when direct references are used, yet the underlying implementation will frequently provide only for indirect references.

Section 5

Design Refinement

5.1 Goals

One of the main goals of the CSS Framework is to support the refinement of an abstract design to a more detailed one. The goal of the Design Refinement task was to demonstrate the use of refinement. In particular, we wished to

- refine the network server specification in the TLS which served as the abstract design,
- explore any issues in refining a design,
- provide an example of refinement analysis, and
- test the refinement portion of the framework.

5.2 Approach

To demonstrate the use of the framework for refinement reasoning, we refined the abstract design for a network server from the TLS. The network server acts as the interface between a kernel and a network. The sole function of the network server is to act as a “forwarding agent” for messages being passed between tasks on the local node and tasks on remote nodes. The operations of the network server implement two high level actions: receive a local message and forward it on to a remote network server, and receive a message from the network and forward it on to a local client.

The network server specification does give a fairly detailed description of the processing required to carry out forwarding operations. However, the network interface is left extremely abstract—the network is viewed as a bag of messages, and message transmission is specified as addition to and removal from the message bag. The next step was to refine this abstract design to a traditional layered protocol stack architecture. We refined message transmission to the level of IP packets and frames being transmitted via an Ethernet medium. We adopted the *x*-Kernel architecture [5] as a model for our protocols; the modularity and clean interface definition of *x*-Kernel protocols work well with the CSS framework and provide a versatile specification paradigm for defining protocol components. In particular,

- The *x*-Kernel separation of protocols is easy to specify in our framework and reinforces the design objectives of composition. In essence, the *x*-Kernel protocol architecture is a compositional framework.
- The flexibility of the *x*-Kernel architecture allows maximal benefits from re-usable analysis. By enforcing strict modularity on its protocols, the *x*-Kernel allows systems to be easily reconfigured, thus providing an opportunity for re-usable components.
- Uniformity of *x*-Kernel communication allows re-usable transition specifications. The transitions that define the open and demux operations in the *x*-Kernel architecture are generic—they are virtually the same for all protocols.

Our *x*-Kernel stack consists of CSS components implementing the following seven protocols:

- **FWD** This component is not strictly a protocol in the sense defined by the *x*-kernel architecture; it has an interface with the lower protocol—DNS—but it does not present the standard interface for communicating with higher protocols (it communicates with a local kernel component.) The FWD protocol is specified in the same format as the lower protocols and for simplicity we include it in our stack definition. The FWD protocol implements the forwarding operations of the network server, but unlike the network server component in the TLS it relies on the lower protocols for transmission of messages across the network.
- **DNS** This protocol specifies a simple domain name server for maintaining bindings between local host names and IP addresses. Our specification does not allow dynamic bindings; rather we use a static look-up table for address translations.
- **DSS** A digital signature protocol for signing messages. The signatures are not visible at the user level; they are used between remote stack instances to ensure message authenticity. We do not specify a particular signature algorithm.
- **DES** A message encryption protocol for maintaining message confidentiality. Again we do not specify a particular algorithm.
- **TCP** A protocol for managing communication sessions, modeled on the standard Transmission Control Protocol as described in the Internic RFC 793. We do not implement the full functionality defined in the RFC standard; in particular we specialize the protocol to handle non-interactive one-way transport of IPC messages.
- **IP** A protocol for implementing best-effort connectionless packet delivery, modeled on the standard described in the Internic RFCs 791, 950, 919 and 922.
- **ETH** This protocol represents a simple Ethernet driver. It maintains bindings between IP addresses and physical addresses; we do not model the acquisition of bindings through the ARP protocol, but rather specify that bindings are contained in a static look-up table.

5.3 Accomplishments

One accomplishment of this task was the demonstration that the CSS Framework supports the specification of an *x*-Kernel protocol stack very well and that the *x*-Kernel architecture is especially appropriate for a component-based specification and analysis style. The clean design of the *x*-Kernel architecture makes a component-based specification particularly convenient. The consistency of interface between the protocol components makes it easy to reuse pieces of the specification and analysis.

We also outlined the proof that the stack is a refinement of the abstract network server. The refinement analysis technique is very valuable for analyzing complex pieces of software such as a network stack. The high-level specification of the network server abstracts out the many details that are needed for realistic network communication, focusing instead on the high-level behavior required of the communication mechanism. This makes it much easier to prove that the overall system of which the network server is only a part has the desired properties. This proof would be much more difficult if the full network stack were included in the analysis. On the other hand, the abstract network server must eventually be implemented and this means dealing with all the complex low-level issues that are necessary to communicate over a network. Refinement analysis allows an analyst to specify the low-level details (to whatever extent is

deemed necessary) and to then show that the resulting network stack is a refinement of the abstract network server. This increases the confidence that the implemented system preserves those characteristics of the abstract design that were essential in proving the desired high-level properties of the system.

5.4 Lessons

One interesting realization that came from this work is that fairness conditions in a high-level specification can never be entirely implemented (i.e., without specifying fairness conditions) at lower levels—there must be some fairness condition at each refinement level. Even though a specific scheduling algorithm might be added at lower levels to define how the high-level actions governed by fairness conditions are scheduled, it will still be necessary to include a fairness condition requiring that the scheduler itself be treated fairly.

The other main lesson deals with the care required in writing the specifications at both the high and low levels to ensure that the refinement is in fact valid. At the high level, one must fight the strong tendency to overspecify the system. There are several advantages that might typically be obtained by keeping the abstract specification as unconstrained as possible:

- It will have greater potential for reuse since there will be more ways to refine it.
- It will be easier to show that a given implementation is a refinement of the abstract specification since there will be fewer constraints that must be demonstrated of the implementation.
- It might even be easier to prove that the abstract specification satisfies the desired critical properties since there will be fewer details that might clutter and obscure the proof.

At the low level, one must be careful not to make tacit assumptions derived from the high level. For example, if one component is split into many we must be sure to specify the allowed interactions between the low-level components so that they cannot do things to each other that the high-level component cannot do to itself. Otherwise, the refinement argument will likely fail.⁹

We also learned that specifying a component in the “simplest” way can sometimes introduce subtle, undesirable constraints.¹⁰ Consider an abstract component c that has two variables, i for input and o for output. At any time it can decide to copy i to o . It does not allow any other changes to o . Now assume that we wish to refine c to consist of a series of network stack components with a network. The variable i corresponds to one end of the communication and the variable o to the other. Each intermediate component performs copy operations. Unfortunately, this is not a valid refinement. At the high level, if o changes, it must change to the value in i . However, at the low level, it may change to the input value of the final network stack component in the chain, and this value need not equal the input value for the first network stack component in the chain. The problem is that the specification for c , although simple, is too constrained. It asserts that c can deal with only one value at a time. This prevents refinement to a low level in which multiple values are in transit. We can make c less constrained by allowing it to keep additional internal state to hold the values in transit. Depending upon the properties needed in c , this state could be a queue, a set or a bag (i.e., a “set” that can contain multiple copies of the same value). The low level discussed above is a refinement of this regardless of which of the three options is selected.

⁹This was also observed in the fault tolerance task.

¹⁰This observation also arises from the fault tolerance proof-of-concept work where the initial specification of the fault-free model was very simple but too constrained to allow the intended refinement.

5.5 Future Work

We had insufficient resources on the program to complete the refinement analysis in PVS. Since the process of doing such proofs (and in particular, machine checked proofs) frequently uncovers errors in the specifications it would be valuable to perform more of this analysis. We expect that more could be learned about the kinds of errors that occur in refining a specification and how those errors might be avoided.

In Section 3.3.1 we discussed impure decomposition of refinement proofs and the support provided by the framework for these decompositions. However, the examples considered on this program all ended up involving pure decompositions. As a test of the theorems for impure decomposition, it would be valuable to work an example that involves impure decomposition.

Section 6

Fault Tolerance

6.1 Goals

The purpose of this task is to demonstrate the use of composition and refinement for the specification and analysis of fault-tolerant systems. This effort included not only specifying new systems but also modifying existing systems to be fault tolerant. The composition framework lets us reason effectively about a system *as a set of components*. Fault-tolerant systems possess certain characteristics that make them well-suited for analysis within a composition framework. For example, potentially faulty components are replicated to achieve fault tolerance. The composition framework lets us reason about one replica and reuse the analysis for the other replicas.¹¹ Fault tolerant systems also have certain unique requirements, such as the need to replicate the inputs to all replicas and to vote across the outputs of all replicas. The composition framework let us model these unique requirements as separate components, which keeps the model flexible. This task also demonstrated how to modify an existing specification (the TLS) to make it fault tolerant.

6.2 Approach

Before describing our approach, it is useful to understand certain facts about fault tolerant systems. A system is *fault-tolerant* if it behaves like a fault free system even in the presence of faults. Our focus is on *hardware* fault tolerance, i.e., the ability of a software system to tolerate faults that originate in hardware. Hardware fault tolerance differs significantly from *software* fault tolerance, i.e., the ability of a software system to tolerate software-induced faults. Butler [3] argues that the standard method for achieving hardware fault tolerance, i.e., component replication, is not effective for software fault tolerance, because the conditions that cause software faults cannot be isolated easily.

Schneider [11] describes two classes of hardware-based faults:

1. *fail-stop*: a faulty component transitions to a state where other components can detect its failure, and then stops, and
2. *Byzantine*: a faulty component acts arbitrarily or even maliciously.

A t fault-tolerant system exhibiting Byzantine failures must have at least $2t + 1$ replicas to ensure a correct vote, while the same system exhibiting fail-stop failures requires only $t + 1$ replicas. According to Schneider [11], every nonfaulty replica should receive every request (AGREEMENT), and every nonfaulty replica should process the requests it receives in the same order (ORDER). AGREEMENT is satisfied when the client transmits the same request to all replicas. ORDER is satisfied when all nonfaulty replicas obey a request sequencing protocol that causes them to process requests in the same order.

The fault tolerance composition study is divided into two phases:

¹¹We did not follow that approach here, because our replicas were very simple.

1. **Proof-of-concept.** Study the specification and verification of fault tolerance properties from a composability perspective.

The proof-of-concept introduces the key concepts for a fault tolerant system: the replication of the potentially faulty component, a replicator component to provide inputs to all of the (potentially faulty) replicas, and a voter component to combine the outputs of the replicas.

These results are presented in *The Fault Tolerance Analysis Report*, CDRL A009 [18].

2. **TLS modification.** Extend the Top-Level Specification (TLS) CDRL A004 [15] to include fault tolerance properties.

The TLS describes four components: an abstract component called the Box Manager and its implementation in three components, the kernel, the network server and the network. Our focus was on the implementation. We assumed that the network server may be faulty, and we proposed a fault-tolerant architecture that replicates the network server and its associated network.

These results are presented in *The Modified Top Level Specification*, CDRL A008 [20].

6.2.1 Proof-of-Concept Approach

In the proof-of-concept phase, we proposed a simple fault tolerant model and demonstrated its validity by postulating a fault free model and proving that the fault tolerant model behaves like the fault free model. The fault free model consists of a single state machine S that accepts requests from its environment and generates responses. State machine S supports three transitions (see Figure 1):

1. If a new request arrives over the `recv` interface, it is placed on the queue `unproc` for unprocessed messages.
2. If `unproc` is not empty, remove the request at the head of `unproc`, process it via

$$\text{process_msg} : \text{message}, \text{proc_state} \rightarrow (\text{message}, \text{proc_state}),$$

place the result (the response) on the processed message queue `proc` and update the processing state. `process_msg` takes a processing state as well as the message because the state may influence the response. In addition to the response, it returns a processing state so that we may consider state machines whose response depends upon the history of past processing.

3. If `proc` is not empty, the head response is removed and transmitted over the `send` interface.

The two queues, `unproc` and `proc`, make possible the refinement to the fault tolerant model which can have multiple messages in progress.

There are few constraints on S or its environment. Requests issued by a single client to S are processed by S in the order they were issued. The interaction between clients is invisible in our model because clients are part of the environment.

We developed an abstract representation of network communication using a shared interface. When component A wishes to send a message m to component B , it places m in the variable `msg` that it shares with B and sets the `done` flag to `FALSE`. Component B recognizes that a new message has arrived when it detects that `done` is no longer true. B extracts m from `msg`

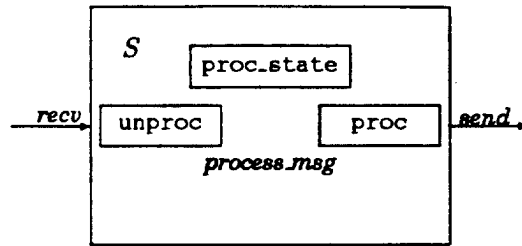


Figure 1: The Fault-Free Model

and sets done to TRUE. The shared interfaces are configured so that messages are transmitted in one direction only. If B needs to reply to A , it uses a different interface. Each component has an interface for outbound messages, called *send*, and an interface for inbound messages, called *recv*, except for the replicator, which has three outbound interfaces, and the voter, which has three inbound interfaces. The *send* interface of a component is equated with the *recv* interface of the component to which it sends messages. The communication model satisfies Schneider's assumption that communication between components is FIFO and nonfaulty, i.e., messages are not randomly created, mangled or lost. Each component specification must constrain how the interface is modified. Since this behavior is similar from component to component, we defined helper functions to simplify the task.

The fault tolerant model (see Figure 2) assumes S may be faulty. It replicates S , yielding $S1$, $S2$ and $S3$, and adds a replicator R and voter V . We introduced a third component, a fault generator, to avoid modifying S to generate faults. We associated a fault generator ($F1$, $F2$ and $F3$, respectively) with each replica to generate any faults that might occur in the replica. The number of fault generators that can be in fault mode is limited by the failure class, i.e., fail-stop or Byzantine. If the failure class is fail-stop a fault generator in fault mode discards all messages. If the failure class is Byzantine, a fault generator in fault mode behaves arbitrarily. If a sufficient number of S replicas agree on their output and if this output matches the output of the single, fault free S for all given request sequences, then the proposed model is fault tolerant.

The relationship between the fault tolerant model and the fault free S is indicated by the dashed box in Figure 2. Note that while R accepts messages for all sending clients, there may be a different V for each receiving client.

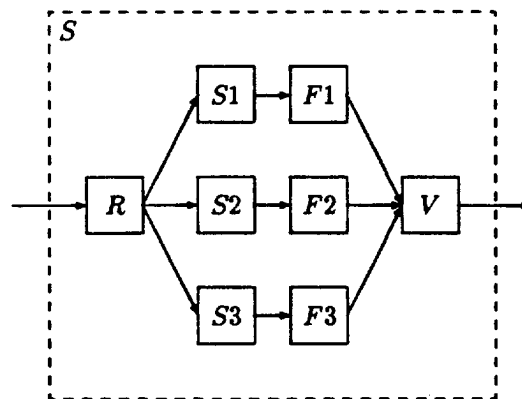


Figure 2: The Fault Tolerant Model

We modeled only three replicas, because three is the smallest number of replicas that make reasoning about Byzantine behavior interesting. We could have avoided specifying the number of replicas since the number of replicas needed is a function of the number of tolerated faults: to tolerate t faults, $t + 1$ replicas are required in the fail-stop case and $2t + 1$ replicas are required in the Byzantine case. Modeling a specific number seemed a simpler task for the proof-of-concept, and it still demonstrates the approach. Our model explicitly assumes $t \leq 1$ for the Byzantine case and it allows as many as two faults for the fail-stop case.

Request ordering is a critical issue for fault tolerant systems. Without it, voting may be impractical. Request ordering is also crucial for the consistency of the internal state of the nonfaulty replicas. Schneider [11] described three protocols that ensure that all replicas process the same sequence of requests. Two protocols are clock-based, while the third lets replicas determine the proper order among themselves. In general, Schneider's protocols are "earliest first". However, the strategy itself does not matter, only that all replicas adopt the *same* strategy. The fault free model assumes nothing about the sequence of requests from S 's environment. S processes the requests in whatever order they arrive. In the fault tolerant model, we had to ensure that replicating S and introducing a new component to handle request dissemination did not affect the order of requests.

Initially, we considered implementing one or more of Schneider's order protocols; however, we discovered that each protocol introduces significant complications for our analysis. The clock-based protocols require that each client attach a timestamp to each request. We could either require this behavior of the client or introduce another component to perform the task on behalf of the client. A similar component would be required at each replica to arrange the requests according to their timestamps. Schneider's third protocol introduces significant inter-replica communication and requires yet another component (different from the clock-based protocols). We realized that additional constraints would be needed *on the fault free model* in order to prove that the fault tolerant model, observing a particular request order protocol, implements the fault free model. For example, if the fault tolerant model ordered requests by timestamps, we would have to constrain S in the fault free model to also order requests by timestamps. As a result, there would exist several fault free/fault tolerant model pairs, each flavored by the chosen request order protocol. To simplify the problem, we avoided implementing any of the protocols.

Instead we adopted a much simpler approach. The requirement for a request ordering protocol is motivated by the presence of *multiple* request replicators, one for each hardware node that supports clients. However, if all client requests are processed by a single replicator, and if that replicator forwards requests in the order that it receives them, then the replicator looks like the single S to the clients. Thus we are guaranteed that the fault tolerant architecture processes the same sequence of requests as the fault free architecture, because the entity that determines the order of requests is the environment, which is the same in both models. We simplified the problem by moving request ordering outside the model and into the environment.

The single replicator approach would not likely be implemented, because the replicator (or rather the hardware node on which it resides) could be a single point of failure; however, for modeling purposes we assume that the replicator is fault free. We realized an important advantage with the single replicator: we get ORDER for free. It is also trivial to demonstrate, given the design of R and the fact that only R transmits requests to the replicas, that AGREEMENT is satisfied.

Note that we solved a slightly different problem than Schneider. He focused on practical implementations for satisfying ORDER. Each of his order protocols includes a *stability test* to determine the next request to process, and he proved that the test is sufficient. We avoided stability tests by letting the environment determine the request sequence.

The trickiest problem for the voter is vote synchronization, i.e., ensuring that the tabulated result is based on a set of votes that are all responses to the same request. Communication delays or other problems may prevent some votes for a particular request from reaching the voter in a timely manner. Since we assume nothing about the vote itself, the voter must rely on other information for synchronization.

A *voting session* occurs when the voter receives a sufficient number of votes (as determined by the failure class) for a particular request. An *obsolete vote* is a vote that misses its intended voting session. The synchronization algorithm is very simple. A counter is associated with each voting replica. When the voter receives enough votes to constitute a voting session, it notes which replica(s) did not vote and increments the counter associated with that replica. When a replica votes and its counter is greater than zero, that vote is discarded and the counter is decremented.

There are several constraints on the fault tolerant model:

- *It must be configured properly.* In other words, the shared interfaces must be assigned between components such that communication occurs only as we require it.
- *All appropriate components must agree on the failure class.* The fault generators and the voter must agree whether the failure class is fail-stop or Byzantine.
- *There may not be too many "active" fault generators for the failure class.* We allow at most two faults in the fail-stop case and only one fault in the Byzantine case.
- *The receive interface and send interface for S in the fault free model must correspond to the receive interface for R and the send interface for V , respectively, in the fault tolerant model.* This constraint is necessary to prove that the fault tolerant model implements the fault free model.

Next, we considered the proof that our fault tolerant model is indeed fault tolerant, i.e., it behaves like a fault free system in the presence of faults. The argument is divided into two obligations:

1. the fault tolerant system only allows the behaviors allowed by the fault free system, and
2. the fault tolerant system allows all behaviors that the fault free system allows.

Obligation 1 is stated formally as

$$\text{implements}(\text{compose}(\{R, S1, S2, S3, F1, F2, F3, V\}), S) \quad (1)$$

That is, every behavior allowed by $\text{compose}(\{R, S1, S2, S3, F1, F2, F3, V\})$ is allowed by S . This proof was broken further into two subproofs:

1. (Initial State) Show that every initial state of $\text{compose}(\{R, S1, S2, S3, F1, F2, F3, V\})$ is allowed by S .
2. (Steps) Show that every transition allowed by $\text{compose}(\{R, S1, S2, S3, F1, F2, F3, V\})$ is allowed by S .

These proofs rely on constraints, supplied by the refinement mapping, that define a state of S corresponding to each state of $\text{compose}(\{R, S1, S2, S3, F1, F2, F3, V\})$. The initial state proof is satisfied easily by relating the external interfaces of the two models and considering the *init* of R and V .

The steps proof is more challenging. The goal of the steps proof is to demonstrate that every state transition allowed by the fault tolerant model is also allowed by the fault free model under the interpretation supplied by the refinement mapping. First we must show that any transition allowed by the *hidd* of every fault tolerant component is allowed by the *hidd* of S . Then we must show that for every fault tolerant component, every *guar* transition (when restricted to the set of transitions allowed by *all* fault tolerant components) is a transition allowed in the fault free model.

Unfortunately, in the course of attempting the formal proofs, we discovered several errors in the specification that prevent the proof from working. These errors are in the form of omitted constraints in the *hidd* of the fault tolerant components. The proofs can be completed if the following missing conditions are satisfied:

- The *unproc*, *proc* and *proc.state* of each S_i component are considered local to that component and can be changed only by agents of that component.
- No agent for a fault tolerant model component can write a message to R 's receive interface.
- Similarly, no agent for a fault tolerant model component can determine that communication over V 's send interface is complete.

These corrections would not be difficult to make in the formal specification; however, there were insufficient resources to do so and complete the proof. Higher priority was given to providing a detailed informal proof in the report [18].

The second obligation is stated formally as

$$\text{implements}(S, \text{compose}(\{R, S1, S2, S3, F1, F2, F3, V\})) \quad (2)$$

In practical terms, the fault tolerance proof requirement stated earlier is unnecessarily strong. Rather than prove that the models are equivalent, it is most interesting to prove that the fault tolerant model does not exhibit behaviors prohibited by the fault free model. This is sufficient to show that critical properties demonstrated for fault free are satisfied by the much more complicated fault tolerant architecture. Other methods, such as testing, can be used to demonstrate that the fault tolerant model exhibits a non-trivial subset of the behaviors permitted by the fault free model.¹²

Finally, we considered the application of our model to a simple example: enforcers and deciders. We argued that enforcers are like the clients of our fault tolerant model, and deciders are valid instances of S . Thus, faults in the deciders could be tolerated by replicating the deciders.

6.2.2 TLS Modification Approach

For the second phase of this task, we modified the architecture described in the TLS to be fault tolerant. We introduced a replicator R and a voter V , and we replicated the potentially faulty NS . The modified architecture is illustrated in Figure 3.

If the client C wishes to send a message to client C' on a different node (kernel), it submits the message on its interface with the kernel K , using its local name for C' . In a fault free system, K would remap C 's local name for C' into a port to which a network server NS is listening.

¹²There is no possible refinement mapping in this direction since multiple fault tolerant states correspond to each fault free state. Thus, a formal proof of this case within the framework is not possible. If quantification were included in the framework, the proof would be possible in principle, but probably very difficult.

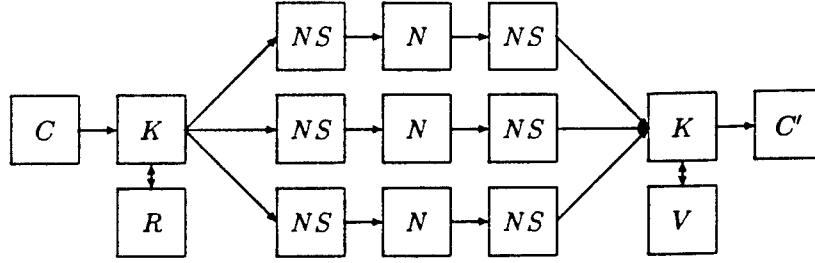


Figure 3: Fault Tolerant TLS Architecture

The *NS* would forward the message over the network *N* to an *NS* on the node shared by *C'*. This *NS* will submit the message to its kernel using its own local name for *C'*. The receiving *K* would place the message on a port to which *C'* is listening. (For a more thorough discussion of communication in the TLS, see the TLS report [15].)

In a fault tolerant system, however, *K* remaps all of *C*'s local names into ports to which the replicator *R* is listening. *R* then copies the message (via *K*) to three different *NS*s. Each *NS* transmits the message over an *N*. At the receiving node, at most three *NS*s receive the message. Each *NS* forwards its message, via the receiving *K*, to a different port to which the voter *V* is listening. *V* determines the proper response and submits the message, via *K*, to the client *C'*.

In the unmodified TLS, clients may also exchange send and receive rights, and the kernel and network servers are responsible for configuring the desired communication path. Unfortunately, the algorithm is complicated significantly by the replication of the network servers and by the introduction of the replicator and voter. Due to the limited scope of this effort, we assumed that the communication paths are in a steady state and that only ordinary data messages are passed between clients in the modified TLS.

Instead of introducing a separate fault generator for each *NS* replica, we modified the network component to exhibit controlled faults.¹³ If a sender-side network server is deemed faulty, the associated network component will exhibit faulty behavior. We assume that all receiver-side network servers are fault free.

Our task was divided into three subtasks:

- Specify *R* and *V* in the context of the TLS.

R and *V* here are slightly different from the corresponding components in the proof-of-concept task, because the communication interface here is more complex. Both components will use the kernel interface for all communication.

- Modify *N* to generate new faults in controlled conditions.

Currently, *N* has two behaviors: it either transmits messages successfully or it randomly loses them. We needed to add a new behavior, the ability to modify a message, and we had to control when these behaviors occur. As in the proof-of-concept task, we designated a subset of *N* components to be faulty according to the failure class (*fail_stop* or *Byzantine*). *N* exhibits faults whenever the associated sender-side *NS* is considered faulty.

- Constrain the global state as required for fault tolerance.

¹³ The network in the unmodified TLS includes faults, but they occur nondeterministically. We needed to ensure that certain networks (actually network servers) are fault free.

6.3 Accomplishments

The proof-of-concept demonstrated that the composition framework is a practical tool for analyzing fault tolerance properties in real systems. For example, we enhanced S to allow faults without changing S ; instead, we composed it with a fault generator. The component specification clearly distinguishes the assertions that a component makes about itself from the assumptions it makes about its environment. Thus, the relationships between components are more obvious. This quality was particularly important for specifying the communication model. The only limitation is that the framework did not support the proof that the fault free model is a refinement of the fault tolerant model, so we demonstrated that result informally. However, this limitation is a problem for refinement mappings generally, not just with the framework. Performing a refinement proof without a refinement mapping requires reasoning directly about infinite sets of infinite sequences, which can be very difficult. Supporting this type of proof would also add significant complexity to the framework, and the dubious benefits do not justify the effort required.

In the proof-of-concept, we simplified the models whenever possible, for example:

- There are exactly three replicas, instead of an arbitrary number.
- The single replicator R is a convenient abstraction for the request order protocols described by Schneider.
- The voter V uses a simple exact-match voting strategy.
- The communication model is based on a shared interface.
- The replicas and S invoke the same, undefined function: `process_msg`. We assume nothing about `process_msg` so that almost any processing function is an instance of it.

We adopted most of these simplifications for the TLS modification task, and we added new simplifications as necessary:

- We assume that only ordinary data messages are passed through the system, i.e., no rights are passed between clients.
- We compromise on a “partially fail-stop” network component. In the existing network component specification, a non-faulty network would never do anything whereas a faulty network could destroy messages. This notion is inconsistent with the concept of a fail-stop system in which a non-faulty component *performs* correct operations and a faulty one does nothing. Rather than change the network component into a more active entity, we leave the possibility that correct data might sometimes be available over a faulty network replica and other times be absent.
- We recognize that we cannot enforce FIFO message delivery in the current network component *as specified*, so we assume that the network component will be *implemented* to enforce the requirement.

6.4 Lessons

This task demonstrated that the composition framework lets us separate concerns in the specification. Specifying the replicator, voter and fault generator as separate components in the proof-of-concept kept the models very general. It also meant fewer changes to the TLS, since

most of the new behaviors could be specified within separate components. Most existing TLS theories were not changed.

We also realized significant savings by letting a single replicator abstract the various message ordering algorithms described by Schneider [11]. As is typical in research projects, we explored many “dark alleys”. Before realizing the opportunity of a single replicator, we spent much time considering which message ordering algorithms to model and exploring modeling approaches. We also considered using one of the communication mechanisms in the TLS (i.e., either the kernel or the box manager) for the proof-of-concept task. However, these mechanisms introduced more overhead than we desired.

6.5 Future Work

While we argued informally in the proof-of-concept task that the fault tolerant model behaves like a fault free model, it would have been nice to complete the proof using the composition framework. It would have also proved interesting to model one of Schneider’s message ordering algorithms. Both the proof-of-concept and the TLS modification could be generalized by allowing an arbitrary number n of replicas. The model would define t as the number of allowed faults, and n would be function of t , according to the failure class.¹⁴ We settled on three replicas because it was not necessary to burden the model with the additional complexity in order to satisfy our goals.

The modified TLS could be improved in the following ways:

- Add support for the passing of send and receive rights.
- Modify the network component to exhibit pure fail-stop behavior and to support FIFO communication.

In subsequent applications of the framework, we have realized additional areas for improvement. For example, the proof-of-concept specification could be generalized by adopting a requirements-oriented, rather than design-oriented, specification approach.

¹⁴For an example of a specification of this style, see the CSS TLS [15].

Policy Composition

7.1 Goals

The policy composition task explored how to incorporate prior work on policy composition into the CSS composition and refinement framework. The goals of this effort were to:

- extend the framework to provide for composable noninterference analysis, and
- study the composition of differing security policies

Some types of security policies can be represented as Abadi-Lamport properties. For example, access control policies such as Bell-LaPadula security can be represented as safety properties asserting that all transitions that occur are consistent with the access control policy. For these policies, the existing CSS framework is sufficient since it provides machinery for reasoning about Abadi-Lamport properties in general. However, one commonly used type of security policy, namely information flow policies, cannot be addressed using Abadi-Lamport properties. The goal of this effort is to investigate how information flow policies can be addressed within the CSS framework.

7.1.1 Information Flow Policies

The difference between an access control policy and an information flow policy is somewhat subtle. The original motivation for information flow policies was to address the covert channel issue in multilevel secure (MLS) systems [8]. A common example of a covert channel is the use of file access times in Unix to signal information. Each file has associated with it an access time indicating the last time the file was read. An MLS version of Unix would check to ensure that the process reading a file is at or above the level of the file before allowing the read to proceed. In the case of a high level subject reading a low-level file, this MLS check would allow the high level subject to change the access time for the low-level file as a side-effect of reading the file. Even though the high-level read of a low-level file is allowed by the Bell LaPadula access control policy, the downward flow of information as a side effect is undesirable.

Information flow policies were developed to address the covert information flows resulting from side effects. In a system satisfying a flow policy, there should be no such covert flows. A variety of such policies have been developed, but we chose to focus on McCullough's Restrictiveness [7] and Rushby's Noninterference [10]. At the heart of both policies is:

- for each system event e , a mapping to a security domain $L(e)$,
- functions *inp* and *out* that test whether events are inputs or outputs,
- a flow relation $d_1 \rightsquigarrow d_2$ indicating when information is permitted to flow from a security domain d_1 to a domain d_2 ,
- for each domain d an equivalence relation \approx_d indicating when two states appear identical from the standpoint of d

From these constructs, security requirements such as the following are constructed:¹⁵

- $inp(e) \wedge (L(e) \not\sim d) \wedge leads_to(st_1, e)(st_2) \Rightarrow st_1 \approx_d st_2$
- $inp(e) \wedge (L(e) \sim d) \wedge leads_to(st_1, e)(st_2) \wedge st_1 \approx_d st_3 \wedge st_1 \approx_{L(e)} st_3$
 $\Rightarrow \exists st_4 : st_2 \approx_d st_4 \wedge leads_to(st_3, e)(st_4)$
- $out(e) \wedge leads_to(st_1, e)(st_2) \wedge st_1 \approx_d st_3$
 $\Rightarrow \exists b : out(b) \wedge \langle e \rangle \approx_d b \wedge st_2 \approx_d st_4 \wedge leads_to(st_3, b)(st_4)$ ¹⁶

Here, $leads_to(st_1, e)(st_2)$ indicates that event e can cause a transition from st_1 to st_2 . The actual security requirements for Restrictiveness and Noninterference vary from the above depending on whether the system specification allows nondeterminism and whether the flow relation is transitive. A system allows nondeterminism if the same input event and starting state can lead to more than one resulting state. A flow relation is transitive if whenever flow is allowed from d_1 to d_2 and from d_2 to d_3 , then flow is also allowed from d_1 to d_3 .

The reason information flow policies are not Abadi-Lamport properties is that Abadi-Lamport properties are simply sets of component behaviors. The above security requirements do not uniquely define a set of component behaviors. Instead they comprise a test that can be performed on a set of behaviors to determine whether the associated system is secure. This has led people to refer to information flow policies as “properties of properties”. In any case, the key point is that they are not Abadi-Lamport properties and hence a different approach is needed to reason about the composition of information flow policies.

7.2 Approach

Restrictiveness has previously been shown to be composable in that:

If each system in a collection S satisfies Restrictiveness, then the composite of the systems in S also satisfies Restrictiveness.

This is a powerful result in that it allows analysis to be done on individual components and then extended to the composite system.

Our general approach for the policy composition work consisted of the following steps:

- Incorporate Restrictiveness into the CSS framework.
- Show that other information flow policies are a special case of Restrictiveness.

Since Restrictiveness is composable with itself, any other policy that is a special case of Restrictiveness can be composed with Restrictiveness. The resulting composite system will satisfy Restrictiveness.

The particular challenges to accomplishing this approach were:

¹⁵Such requirements are commonly referred to as unwinding conditions for the flow policy rather than the flow policy itself. Usually the flow policy itself requires that equivalent event sequences lead to equivalent states. The unwinding conditions reduce such policy statements about sequences of events to statements about individual events. Performing analysis with respect to the unwinding conditions is generally much easier than performing analysis with respect to the flow policy itself.

¹⁶Here, b is an event sequence and out and $leads_to$ are the obvious extensions from functions on events to functions on event sequences. The application of \approx_d to event sequences indicates whether each flow contains the same set of allowed flows.

- The type of composition with respect to which Restrictiveness is composable is different than the CSS notion of composition.
- Restrictiveness assumes a transitive flow relation. Thus, generalizations might be required in Restrictiveness to ensure intransitive information flow policies are a special case of Restrictiveness.

7.2.1 Detailed Approach

To extend the framework, we defined a *system* to be a structure with fields:

- *cmp* — a CSS component,
- *inp* — a set of agents denoting events that are system inputs,
- *vws* — a mapping from security domains to equivalence relations on states,
- *D* — a relation indicating whether flow is allowed from d_1 to d_2 , and
- *L* — a mapping from agents to security domains.¹⁷

Since information flow policies are generally stated in terms of events and states, we needed a way to represent events within the CSS framework. Events in information flow policies are used to label state transitions, while the CSS framework labels state transitions with agents. Equating events with agents is the natural approach to use in incorporating information flow policies into the framework.

With these constructs, the formalization of Restrictiveness for *systems* was simply a matter of translating the formalization in the literature. We next attempted to demonstrate that Restrictiveness is composable with respect to the framework definition of composition. As mentioned earlier, the CSS notion of composition is different than the type of composition with respect to which Restrictiveness has previously been shown to be composable. The primary difference is that the other type of composition assumes that systems have disjoint state spaces. The motivation for this is that events represent messages passed between loosely coupled systems. In contrast, agents in the CSS framework are more to denote responsibility for a transition than to denote information being communicated; communication is assumed to occur through shared state information.

The significance of this difference poses a problem when showing that security requirements of the form illustrated in Section 7.1.1 are composable. Then, the requirements are assumed to hold for each of a collection of systems and the goal is to show the composite of those systems satisfies the requirements. Some of the requirements assert the existence of a st_4 satisfying certain properties. Unfortunately, the fact each individual system satisfies the requirements only guarantees the existence of a set of st_4 's each of which satisfies the desired properties for one of the individual systems. To prove composability, a st_4 must be exhibited that satisfies the properties for the composite system. Essentially, this requires exhibiting a single st_4 that satisfies the necessary properties for each system. In Restrictiveness' version of composition, the desired st_4 is simply the merger of the st_4 's known to exist for each system. Since the state spaces are independent, this merger is well-defined. When the state spaces are not independent,

¹⁷In the PVS itself, we refer to "security domains" as levels. This is an artifact of the work initially being based on MLS policies. Also note that *D* and *L* are actually generic parameters to the *system* theory rather than fields of the *system* structure. The only real significance to this distinction is that *D* and *L* must be static throughout the execution of a system.

some way is needed to merge the st_4 's known to exist for the individual components into a st_4 acceptable to all components.

The only solution we developed to this type of problem was to add assumptions about the set of systems being composed. Thus, the composition theorem was of the form:

If sys_set satisfies certain hypotheses and each element of sys_set is Restrictive, then the composition of the elements in sys_set is Restrictive.

This is a less desirable composition result than that for Restrictiveness since the additional hypotheses must be considered each time Restrictiveness of components is to be lifted to Restrictiveness of the composite. Note, however, that when the Restrictiveness' simplifying assumption about state space independence is made, the proof of the additional hypotheses is trivial. This simplification is only possible when specifications are written in a loosely coupled manner. For more tightly coupled components, the additional hypotheses are a concern, but then the original form of Restrictiveness cannot even be used.

As an alternative, we also considered a strictly stronger policy definition based on *event closure*. A system is said to satisfy *event closure* if:

$$\begin{aligned} & \blacksquare \text{ leads_to}(st_1, e)(st_2) \wedge st_1 \approx_{L(e)} st_3 \wedge st_2 \approx_{L(e)} st_4 \\ & \Rightarrow \text{leads_to}(st_3, e)(st_4) \end{aligned}$$

In other words, whenever (st_1, st_2, e) and (st_3, st_4, e) denote equivalent transitions with respect to $\approx_{L(e)}$, then if one transition is valid in the system the other transition must be valid, too.

We define a *System*¹⁸ to be a *system* that satisfies the unwinding condition dealing with "high" inputs¹⁹ (known as the Local Respect condition in Rushby's work) and satisfies event closure. We showed the conjunction of these conditions is composable. This means that a composition of *Systems* is itself a *System*. Event closure implies the unwinding condition dealing with "low inputs" (known as the Step Consistency condition in Rushby's work). So, the fact that composition preserves *Systems* results in a policy that is very similar to Restrictiveness yet composable using our definition of composition. If event closure is an acceptable policy for an analyst, the analyst could use event closure and not need to prove additional hypotheses hold when lifting event closure of components to a composite system. However, neither the Local Respect condition nor event closure subsumes Restrictiveness' requirement on outputs and consequently they alone might not be sufficient security requirements.

A third option explored in this work is Restrictive *Systems*—that is, systems that satisfy all the requirements of McCullough restrictiveness and satisfy event closure. This definition of security is strictly stronger than McCullough Restrictiveness. The associated composability theorem for Restrictive *Systems* has fewer proof obligations than does the theorem for restrictive *systems*. Thus, this approach represents a compromise between the first two approaches.

Next, we demonstrated that Rushby's transitive noninterference is simply a deterministic version of Restrictiveness. This ensures that as long as each component system is either restrictive or noninterfering, then the composite is restrictive as long as the additional hypotheses hold.

Finally, we investigated the generalization of Restrictiveness to incorporate intransitive policies. Although it was relatively straightforward to incorporate Rushby's concepts for intransitive policies into the definition of Restrictiveness, we were unable to demonstrate the resulting

¹⁸Note the capitalization.

¹⁹By a high input, we mean an input event whose security domain is not permitted to communicate with the domain with respect to which the view is being taken.

generalization was composable. We were successful in demonstrating that the Local Respect and Step Consistency conditions are composable (once again under some additional hypotheses on the components being composed). However, we were not able to complete the proof that the unwinding condition dealing with system outputs is composable before running out of time on the program. This proof is the hardest part of proving the unwinding conditions are composable, so we really made little progress on demonstrating the generalization for intransitive policies is composable.

Although most applications of information flow policies in practice have been with respect to transitive (MLS) policies, having theories of intransitive policies would be useful. Example applications include the following:

- Consider an MLS system that contains certain trusted processes that have privileges allowing them to downgrade information. It is not possible to prove a transitive noninterference policy because the downgrading of information would violate the policy. However, an intransitive flow relation could be defined that allows:
 - any information flow upward in level,
 - any information flow to a trusted process, and
 - any information flow from a trusted process.

Rushby's intransitive noninterference with this flow relation would require that information only flow downward in level if it goes through a trusted process. Analysis with respect to this policy would identify any downgrades that bypass the trusted processes.

- Consider an *assured pipeline*. In other words, consider a system that requires information to flow through certain stages. As a specific example, consider a firewall proxy protecting internal systems from the internet. An intransitive flow relation could be defined that allows:
 - any information flow from the internet to the proxy, and
 - any information flow from the proxy to the internal systems.

Analysis with respect to this system could detect flaws that allow the proxy to be bypassed.

The naturalness of intransitive policies to state such real-world policies is a strong argument for developing theories to support intransitive policies.

7.3 Accomplishments

The positive results of the program are:

- We were able to formalize information flow policies within the framework.
- We were able to prove some composability results for these flow policies.
- We demonstrated transitive noninterference is a special case of Restrictiveness.
- We developed a generalization of Restrictiveness that subsumes intransitive noninterference. We also completed small parts of the proof of composability of this definition.

- We hypothesized that any information flow policy can be partitioned into a purely transitive information flow policy plus a purely intransitive information flow policy. This would provide a nice “normal form” for policies. We sketched a proof that this decomposition is always possible, but did not spend much time verifying the details of the proof.

Limitations of the results of the program are:

- We were unable to complete the composability proof for the generalization of Restrictiveness. This means that our composition result does not currently support intransitive policies. We do not yet know whether the desired result is unprovable or whether additional time would allow us to complete the proof. We have, however, noted the need to change the proof strategy based on differences in how equivalent event sequences are defined in the transitive and intransitive cases.

Sequence equivalence is defined in terms of event purging which removes from the sequence those events that are prohibited from communicating with the domain of interest. In the transitive case, the purgeability of an event depends on only the label of the event. In contrast, purgeability in the intransitive case depends on what events occur later in the sequence. To clarify this distinction, suppose flow is allowed from A to B and from B to C , and events a and b have labels A and B . Event a is purgeable with respect to C in the sequence $\langle a \rangle$ since there is no path from A to C in the remainder of the sequence. However, since a is a path from A to B and b is a path from B to C , a is nonpurgeable with respect to C in the sequence $\langle a, b \rangle$.

The significance of this difference is in piecing together an acceptable output sequence b for the composability proof for the condition on system outputs. The proof proceeds by induction and considers a sequence $ag \circ a$ in the inductive step. Since the requirement on outputs is assumed to hold for individual systems, it is possible to obtain a sequence b_1 that is equivalent to $\langle ag \rangle$. The inductive hypothesis can be used to obtain a sequence b_2 that is equivalent to a . In the transitive case, $b_1 \circ b_2$ can be used for b . The context insensitivity of purging means that concatenation of equivalent sequences results in equivalent sequences. The failure of this property to hold for intransitive purging means a different strategy must be used for the proof.

- Our restrictiveness composition result contains additional hypotheses on the systems being composed. This is undesirable because it requires additional analysis to be performed whenever there is a need to lift the Restrictiveness of individual components to the Restrictiveness of the composite. The additional analysis should be trivial in cases when McCullough Restrictiveness is applicable. So, our formulation essentially contains McCullough Restrictiveness as a special case.
- Our event closure composition result does not contain additional hypotheses, but is strictly incomparable to Restrictiveness. The event closure property itself is strictly stronger than Restrictiveness' Step Consistency. However, Restrictiveness' requirement on outputs is not addressed by event closure. This makes it unclear whether event closure and the Local Respect conditions alone are adequate properties for analyzing systems.
- The Restrictive *System* result is probably the best of the three composability results. Its definition of security is stronger than that of McCullough and its proof obligations are easier than our first restrictiveness composition result.
- We have no practical experience using our composition results. We do not know, for example, how difficult it is to justify the additional hypotheses present in the composition theorems.

- Finally, we considered only two forms of information flow policies. There are a variety of other types of information flow policies. At present we have no idea how composable other types of flow policies are with Restrictiveness and Noninterference. In retrospect, we underestimated the amount of time required for this task and thus did not budget enough time to consider a wide enough variety of policies.

7.4 Lessons

All lessons learned have been captured above in Section 7.3.

7.5 Future Work

Much work remains to be done in this area. First, we need to sample more of the types of information flow policies in existence and give consideration to how they might be composed. If many of these policies are not easily subsumed under Restrictiveness, then the overall approach of developing a general, composable information flow policy must be abandoned in favor of a calculus indicating for each pair of policy types the result of a composition.

Regardless of which approach is found best, there still remains much to do for each. For the unifying policy approach, the general, composable policy must be developed and shown to subsume the various existing policies. In addition, experience must be gained regarding how difficult the additional hypotheses in the composition theorem are to justify. If the calculus approach is pursued, then the various policies must be formalized in the extended CSS framework and composition theorems must be proved for each pair of policy type. Once again, practical experience using the resulting composition theorems is needed.

Section 8

Tool Support

8.1 Goals

The goal of the tools task was to identify and prototype automated software tools to assist in the design and refinement process including the proofs involved in applying the framework. Preference was given to adapting existing tools, rather than creating new ones. Two primary tools were identified.

- **Specification Browser**

This tool assists in writing and maintaining specifications in the form required by the framework. The Specification Browser also aides analysts and developers in achieving and maintaining consistent coverage in formal and informal descriptions of a software component.

- **Analyst's Assistant**

This tool facilitates the use of the framework by analysts new to PVS and to the framework. The assistance provided comes in two forms:

- When possible, automate portions of proofs.
- Provide analysis hints on the proof approach to use and help the analyst construct lemmas facilitating the analysis.

We discuss each of these tools in this section.

8.2 Approach

8.2.1 Specification Browser

The CSS methodology encourages parallel refinement of the formal model and the system design to a detailed level. This produces a stronger link between the assurance analysis and the detailed design, resulting in a higher level of assurance that the developed code correctly implements the specification. The Specification Browser supports the CSS methodology by combining the formal specification and the design description in the same document. This helps code developers and assurance engineers provide consistent coverage in text description and formal specifications, strengthening the connection between assurance and design through successive refinement steps. Because code developers and assurance engineers work from the same information, there is an increased level of assurance. Also, the Specification Browser guides specifiers in supplying all the sections required for use in the composability and refinement frameworks.

To achieve the goals of parallel refinement, close correspondence and ease of writing specifications, the Specification Browser was designed with the following functionality in mind:

- The browser has an understanding of the structure of a specification written for the framework. This includes

- the kinds of information commonly included in a formal specification (e.g., initial state requirements and transitions)
 - convenient ways to structure this information for use in a specification in the CSS Framework.
- The browser uses this understanding to instantiate templates for documents, sections, transitions, etc. into which specifiers can type the information for the component being specified. The browser helps link these templates together to form a complete specification document.
 - The default structure of a specification document will keep the formal and informal descriptions as close together as possible to encourage cross-checking for consistency.
 - The browser provides cross-referencing and search functionality that helps relate the informal and formal specifications. For example, both versions can be displayed simultaneously, and regular expression searches can be performed in the formal and informal specifications to help correlate definitions of concepts.
 - Since it is frequently useful to have a tabular representation of the behavior of a component or a piece thereof, the browser supports case coverage (processing) tables.
 - To allow specifiers flexibility in the final appearance and structure of their documents, the order of sections is configurable.
 - The browser provides a high-level view of the document to aide navigation through a large specification.

Specification documents produced with the browser intermingle formal specifications and text descriptions of each component that is specified. Each formal specification has a corresponding text description. Several tools are used to create these specification documents: PVS, Emacs, the Specification Browser, \LaTeX , and tcl/tk. The relationship of some of these tools is shown in Figure 4. The formal specifications are written with PVS (Prototype Verification System). The Emacs display editor provides the interface to PVS. Although text descriptions may be written using any text editor, Emacs can be customized or extended using the built-in Emacs LISP language. Emacs also provides some support for producing \LaTeX documents. The Specification Browser is implemented using Emacs as a front-end user interface. The common interface with PVS, the built-in support for \LaTeX , and the built-in Emacs-LISP language contributed to this decision.

The Specification Browser makes use of several template files. These files help by identifying all the required and optional sections for specification documents that follow the CSS methodology and by providing the “boiler plate” information and driver files necessary to produce a \LaTeX document.

The processing table tool is written in tcl/tk. It provides a graphical interface for creating processing tables that describe the return value and state changes associated with each value of the relevant Boolean conditions on the input parameters and current state. The processing table tool can also be used to verify that all combinations of Boolean values are covered by a given table. This tool was originally developed on another program and has been adapted to work with the Specification Browser.

Understanding the high-level structure of a specification means that the Specification Browser can determine in which specification section the point of the current Emacs buffer is located and how that section relates to others in the document. This understanding also allows the Specification Browser to find and simultaneously display corresponding formal and informal

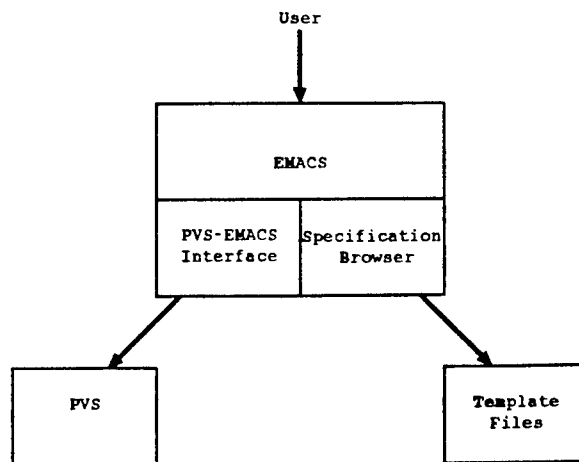


Figure 4: System-level diagram.

descriptions. Also, the Specification Browser will still function if a user changes the order of specification sections or decides to omit certain sections although some of the features may cease to work.

To implement this structure awareness, the Specification Browser uses formatted \LaTeX comments inserted in the specification's text sections to uniquely identify each specification section. The comments do not affect the output of the specification document. The comments are of the form

```
*TLS%unique identifying string%
```

Each formatted comment indicates the start or end of a specification section of a particular type. There are formatted comments at the start and end of each template file which help to identify that file. For example, a file may contain a component specification or a process transition.

8.2.2 Analyst's Assistant

Early in the effort, we decided the most reasonable approach to explore was the construction of PVS strategies to simplify the proof effort. PVS strategies are essentially LISP programs that can be executed from within the prover to access and manipulate the current state of the theorem being proved. The underlying PVS prover ensures any manipulations of the theorem are logically sound. The developers of PVS provide the strategy facility to allow others to extend the PVS prover as they see fit. This is the natural approach to providing analysts automated support within the prover.

Unfortunately, there is little documentation currently available on how to construct interesting strategies. In particular, little information is available on how to access the internal structures representing a theorem. Without this information, proof steps cannot be tailored to fit the form of the current theorem. Part of this effort, consequently, was aimed at reverse engineering the internal data structures used to represent theorems. The LISP command `describe` was the key to this part of the effort. It displays essentially all there is to know about any data structure provided to it. By starting with the variable holding the current state of the theorem, we were able to repeatedly use `describe` to pick the structure apart. Examples of the types of things we were able to do include:

- Test PVS expressions to see if they satisfied a specific form. For example, the framework function *comp_1* denotes the set of restrictions on components. For each component specified, a goal of the form:

- *comp_1(cmp)*

is generated. Once a method is devised to access the internal state, it is possible to test whether the current theorem is of a given form, for example, *comp_1(.)*. This is a handy technique for deciding what type of requirement is being proved and which proof steps are appropriate.

- Look up function definitions in the database PVS builds when it parses a specification. As an example application, consider expanding the definition of a component. Typically, we specify functions such as *cmp_spec*, *cmp_spec_base*, *cmp_init*, and so on. Upon finding *cmp_spec* in the statement of the theorem, it is useful to be able to look up its definition and find the need to expand *cmp_spec_base*. Similarly, it is useful to be able to look up the definition of *cmp_spec_base* and identify the need to expand *cmp_init*. Once the list of functions to expand is identified, a proof command can be generated and executed that expands exactly those functions.
- Traverse PVS expressions. For example, given an expression such as $x = A \vee x = B \vee x = C$ we were able to build a function that returned a list of the possible elements for x (in this case, (A B C)).

In general, our approach was to construct a single strategy for use by analysts. This strategy is called *css-prove*. The general structure of the strategy is:

```

if theorem is of class 1, then execute strategy 1
else if theorem is of class 2, then execute strategy 2
...
else if theorem is of class n, then execute strategy n
else indicate no help can be provided

```

In other words, *css-prove* recognizes certain classes of theorems and for each has an associated proof script intended to automate some or all of the proof. This task was executed as level-of-effort so we simply continued adding new classes to *css-prove* until the budget was expended. In addition to associating proof scripts with each class of theorem, we also associated a help page with each class. The help page provides a description of the class, suggests a general approach to performing the proof, and indicates theorems and definitions from the framework that might be of use in the proof. If the analyst sets the *:help* flag to *t* when calling *css-prove*, then rather than the class' proof script being executed *css-prove* simply displays the appropriate help page.

The intent of this approach is to allow the analyst to use the *css-prove* command whenever he is unsure how to proceed. If the current theorem is one of the recognized classes, then the strategy can start the proof in the correct direction (and sometimes complete the proof itself) or provide the analyst with help. For analysts that know little about the framework or PVS, having a single command that helps them learn about both should be very valuable. Even for analysts that are knowledgeable of both the framework and PVS, the strategy can sometimes save the analyst the trouble of performing a tedious proof manually.

8.2.2.1 Detailed Approach We now provide an example of what we mean by a theorem class, how `css-prove` recognizes a theorem class, and what a proof script and help page look like. For this example, we will consider theorems of the form:

```
impl_init(compose(stack_set(ns)), net_serv_comp(ns))
```

This is an instance of a theorem class. The class is parameterized by a set of implementation (low-level) components and a specification (high-level) component. In other words, the class has the following template:

```
impl_init(compose(_), _)
```

The first step in implementing the proof strategy is to recognize prover states that match this template. To do so we search the consequent (the things to be proven) in the current prover goal looking for a formula that satisfies the following:

- it is an application of the `impl_init` function,
- it has two arguments, and
- its first argument is an application of the `compose` function.

The next step is to develop a sequence of prover commands to be returned by the strategy for execution by PVS. This proof strategy is one of the more complicated ones developed under this task and consequently provides a good illustration of the range of strategies that can be developed. Note, however, that this particular type of proof is too difficult to completely automate. Instead, this strategy simply automates the common steps that typically need to be taken at the beginning of the proof. The hope is that by getting the analyst started in the right direction, the analyst can more easily complete the proof.

The definition of `impl_init(compose(set), cmp)` is simply that `init(compose(set))` is contained in `init(cmp)`. The definition of `compose` is such that `init(compose(set))` is the intersection of `init(c)` for each `c` in `set`. So, letting c_1, \dots, c_n denote the elements of `set`, it suffices to show that:

$$init(c_1)(elem) \wedge init(c_2)(elem) \wedge \dots \wedge init(c_n)(elem) \Rightarrow init(cmp)(elem) \quad (3)$$

The strategy we defined reduces the initial goal to this simpler goal. In addition, it expands the definition of each component until the function defining the `init` for the component is reached. For example, if f_1, \dots, f_n denote the functions defining `init` for the c_i 's and f denotes the function defining `cmp`'s `init`, then the strategy reduces the goal to:

$$f_1(elem) \dots \wedge f_n(elem) \Rightarrow f(elem)$$

Having the strategy perform additional proof steps such as expanding the definitions of the f_i 's and f or perform a `grind`²⁰ would be trivial. However, the one sample theorem of this form that we attempted was not provable by simply grinding. We felt that the expansion of additional functions left the theorem in too complex of a state to be a useful starting point for the analyst and chose to stop at this point. This trade between automating as much as possible and providing the analyst an understandable starting point is an issue for any proofs that cannot be completely automated.

Now that we have described our intent for the strategy, we can describe its implementation. The first step we take is to show that:

²⁰The `grind` strategy is a powerful strategy provided with PVS that repeatedly expands function definitions and performs logical rewrites. This strategy is powerful enough to automatically prove simple types of theorems.

$$\text{init}(\text{compose}(\text{set})) = \{ \text{elem} \mid \text{init}(c_1)(\text{elem}) \wedge \dots \wedge \text{init}(c_n)(\text{elem}) \}$$

We do this by constructing the negation of the above expression and using it for a case split. The first case of the proof requires proving the expression true and can be done using a grind. In the second case, we can assume this expression holds. After expanding the appropriate functions, we see we can assume that $\text{init}(\text{compose}(\text{set}))(\text{elem})$ holds for some elem . Using the expression proven in the first case, we can assume elem is an element of each $\text{init}(c_i)$. Then, all that remains for the strategy to do is expand the c_i 's to get to the f_i 's and expand cmp to get to f .

The hardest part of the strategy is constructing the expression upon which to case split. By picking apart the conclusion, an expression representing set can be extracted. Using suitable accessor functions discovered using `describe`, the definition of set can be extracted. This strategy assumes that the definition is of the form:

$$\{c \mid c = c_1 \vee \dots \vee c = c_n\}$$

PVS represents such an expression using a SET-EXPR data structure having accessors BINDINGS and EXPRESSION. In this case, BINDINGS returns a singleton list containing c and expression returns $c = c_1 \vee \dots \vee c = c_n$. PVS represents this expression using an INFIX-APPLICATION structure with operator OR and arguments $c = c_1$ and $c = c_2 \vee \dots \vee c = c_n$. A recursive function was written to traverse such an expression and return the list c_1, \dots, c_n . Another recursive function was written that took such a list and converted it to an expression of the form $\text{init}(\text{compose}(\text{set}))(\text{elem}) \Rightarrow \text{init}(c_1) \wedge \dots \wedge \text{init}(c_n)$.

The strategy uses the PVS strategy branch to split the proof into cases. Branch takes the form:

```
(branch (cmd) ((cmd1) (cmd2) ... (cmdm)))
```

In this construct, cmd , cmd1 , ... cmdm are each strategies themselves. The strategy cmd is assumed to break the current goal into m cases. The strategy cmdi is applied to case i . For the strategy being built here cmd is the case split, cmd1 is a sequence of commands for proving (3), and cmd2 expands impl_init , subset? , the c_i 's and cmp .

Most of the strategies constructed on this task are much simpler. Generally, they simply identify the class of theorem and perform a restricted grind to expand an appropriate set of functions. This example shows, however, that more complicated strategies can easily be constructed. See the Tools Report for more details on this and other strategies.

The associated help page for this theorem class is as follows.

The current goal is of the form $\text{impl_init}(\text{compose}(\text{set}), \text{cmp})$. The definition of impl_init requires that any element belonging to the init of the first component, $\text{compose}(\text{set})$, is an element of the init of the second component, cmp . The definition of init for a composite is the intersection of the init for each component composed. Thus, the first step of the proof is to reduce the goal to showing any element belonging to init for every element of set is in $\text{init}(\text{cmp})$. Then, it is necessary to show that the combined requirements of the init 's for the elements of set are sufficient to establish the requirements of init for cmp . The strategy is aimed at automatically handling the first step of the proof. Experimentation with the second step of the proof shows it is not easily automated, so the default is for the analyst to handle the second step himself.

The everything flag controls whether the default behavior occurs. If set to nil, the default occurs. Otherwise, a grind command is executed in an attempt to complete

the proof. In experiments tried, this grind command can take a long time and still fail to complete the proof.

8.2.2.2 A More General Strategy The `css-prove` strategy provides focused help on the recognized theorem classes but provides no help for other types of theorems. To provide more general proof support, we defined a general strategy called `css-stewn`. The PVS provided `stew` strategy allows the analyst to specify a set of definitions and theorems to use in the proof and then makes use of those hints to try to automatically prove the theorem. For simple theorems, this strategy can be quite effective. However, for more complicated theorems, great care must be taken in specifying the hints. Generally the approach used is to first expand all definitions and then try to use logical operations to complete the proof. If too many definitions are provided as hints, then the strategy will expand much more than it needs. Even with moderately complex theorems we have seen `stew` take hours to complete. On the other hand, if the analyst does not specify enough definitions as hints, then the strategy might not be able to complete the proof. The `stew` strategy does provide methods to fine tune the hints such as specifying definitions that should not be expanded, but striking a proper balance can still sometimes be challenging.

The `css-stewn` strategy is simply a front-end to `stew`. It allows the analyst to provide some additional hints that are used to fine tune the parameters provided to `stew`. The simplest type of fine tuning enabled by `css-stewn` is the specification of an expansion level. Specifying level 0 results in no definitions being expanded beyond those explicitly listed by the analyst. Specifying level 1 causes the functions appearing in the current theorem to be added to the list. Specifying level 2 causes the functions used in the definition of the functions appearing in the current theorem to be added. In other word, each time the level is increased, function expansion is done to one level lower. This provides a simple way for the analyst to prevent expansion from being done too deeply.

For analysts who have knowledge of the internal representation of the theorem as well as some LISP programming experience, more advanced hints can be provided through `css-stewn`. This is accomplished by allowing the analyst to specify a LISP function defining a stopping criterion for expansion. This allows the analyst to request functions be expanded until the theorem has a certain form. For example, the analyst could instruct the strategy to expand functions until the current goal is a universally quantified expression. As another example, the analyst could indicate that definitions should be expanded until the current goal is of the form $A = B$. Given the need to understand the internal structure of theorems and LISP programming, this advanced feature of the strategy is expected to be of use to strategy developers rather than analysts. In fact, we expect the current definition of `css-prove` could be simplified by using `css-stewn`. Unfortunately, `css-stewn` was the last strategy we developed so is not used by any of the other work.

8.3 Accomplishments

8.3.1 Specification Browser

The Specification Browser prototype implementation achieved most of its goals. It provides an automated environment with some flexibility for creating specification documents that follow the CSS methodology. Because of its understanding of the high-level structure of a specification, the Specification Browser supports reviewing corresponding formal specifications and text descriptions. The Specification Browser user interface shows an outline of the specification document based on the required and optional sections for use with the composability and refinement frameworks. Regular expression search was implemented and the processing table

tool was integrated. The template files reduce the overhead associated with producing a \LaTeX document. Since the browser was implemented in Emacs LISP there is the potential for significant interaction with PVS functionality, but this has not been exploited in the prototype.

There are some limitations in the browser functionality:

- Configurability of section order was only partially achieved. This turned out to be a difficult requirement. Currently, the formatted comments which identify whole sections of the specification document can be placed in any order within a single component specification or transition file. However, the formatted comments are insufficient to connect a PVS file to a text section unless the PVS file's formatted comment appears in that text file. As a result, if someone decided to place all the PVS theories in an appendix, other Specification Browser features such as viewing corresponding sections, checking specification files, and the Specification Browser outline interface would not operate properly.
- The output of regular expression searches could be more helpful. A regular expression search can be used from the Specification Browser outline interface and from any file that is part of the current specification document. When a search is performed, the results are displayed in a buffer showing the files and locations where matches are found. This buffer may be used to move directly to the buffer and the location of the found regular expression. The feature could be improved if the search results screen also showed the string found by the regular expression search. Showing the line that contains the found regular expression might be even more helpful. Also, it may be helpful to mark the search results after having visited one of the found regular expression. This would identify which matches the user has already visited.
- The browser is probably of greatest help to someone inexperienced with the framework and with \LaTeX . It might not improve efficiency of someone who is experienced in producing specification documents of the type produced by the browser.

8.3.2 Analyst's Assistant

The following proof classes are currently recognized by `css-prove`:

- `satisfies(-, always(-))`, where the second parameter is an arbitrary state or action predicate

`assum_satisfies(-, always(-))`, where the second parameter is a state or action predicate

The strategy uses theorems in the framework to reduce these goals to simpler goals. The analyst is expected to provide names of lemmas asserting these goals hold.²¹ If appropriate lemmas are provided as hints, the proof is automatically completed. Otherwise, the strategy displays what the lemmas should look like and leaves the reduced goals for the analyst to prove.

- `steps_satisfy(-, stable(-))` or `assum_steps_satisfy(-, stable(-))`

The strategy expands the framework functions defining satisfaction and stability as well as the top level functions defining the component and state predicate of interest. This saves the analyst the hassle of performing certain common steps that must always be performed for this type of proof, however, the analyst must complete the proof manually.

²¹ Actually, the strategy defaults to using variants of the current theorem name as the lemma names. By following a specific naming convention, this saves the analyst the trouble of providing the hints.

- *steps_satisfy*(-, -) or *assum_steps_satisfy*(-, -) where the second parameter is an arbitrary action predicate

The strategy expands the framework functions defining satisfaction as well as the top-level functions defining the component and action predicate of interest. This saves the analyst the hassle of performing certain common steps that must always be performed for this type of proof, however, the analyst must complete the proof manually.

- *init_satisfies*(-, -)

The strategy expands the functions defining *init_satisfies* as well as the top-level functions defining the component and state predicate of interest. The analyst must complete the proof manually.

- *init_restriction*(-)

The strategy reduces the goal to demonstrating the existence of a state satisfying the *init* for the component of interest. The analyst can optionally provide as a hint a lemma asserting the existence of such an element. The proof is completed automatically if the hint is provided.

- *cags_restriction*(-)

The strategy reduces the goal to demonstrating the existence of an agent satisfying the *cags* for the component of interest. The analyst can optionally provide as a hint an element believed to be in *cags*. If the hint is provided the PVS provided *grind* strategy is used to attempt to complete the proof by showing the specified element is in *cags*. Our experience is that when the hint is provided the proof can typically be completed automatically.

- *guar_restriction*(-), *rely_restriction*(-), *hidd_restriction*(-), *view_rely_restriction*(-), *view_hidd_restriction*(-), *view_guar_restriction*(-), *view_init_restriction*(-), *view_wfar_restriction*(-), *view_sfar_restriction*(-), *guar_stuttering_restriction*(-), *rely_stuttering_restriction*(-), *hidd_stuttering_restriction*(-), *wfar_restriction*(-), *sfar_restriction*(-), *wfar_stuttering_restriction*(-), or *sfar_stuttering_restriction*(-)

The strategy expands the functions defining the restriction of interest as well as top-level functions defining the component of interest. In some cases, this is sufficient to complete the proof. Generally, though, additional functions need to be expanded to complete the proof. These additional functions can be provided as hints to the strategy. Then, the proof is completed automatically. The functions that must be provided as hints are usually straight-forward to determine from the structure of the specification.²²

- *comp.t*(-)

The strategy expands *comp.t*. This reduces the goal to showing each of the component restrictions holds. Since the component restriction theorem classes were addressed above (see the preceding three bullets), this theorem class will not be discussed further here.

- *VIEWS*(vw)

The strategy simply executes the *grind* strategy. Due to the way in which CSS specifications are written, this was found to always complete the proof automatically.

- *init*(-)(-)

The strategy simply executes the *grind* strategy. With the CSS specifications, this was always found to automatically complete the proof.

²² An obvious enhancement would be to have the strategy identify these functions automatically.

- *impl_init(compose(-), -)*

This theorem class was used in the previous section to illustrate the strategy development process. The strategy reduces the goal by expanding framework functions and the definition of *init* for the composite, but the analyst must complete the proof manually.

- *subset?(guar(compose(-)), guar(-))* or *subset?(hidd(compose(-)), hidd(-))*

The strategy operates on these classes in a manner analogous to that for *impl_init*.

- *impl_steps(compose(-), -)* or *implements(-, -)*

The strategy expands *impl_steps* or *implements* to reduce the goal to demonstrating previously described theorem classes. The analyst can then invoke the strategy again to work on each of the generated goals. Alternatively, the analyst can provide lemma names addressing the generated goals in which case the proof is completed automatically. The strategy defaults to trying variants of the theorem's name as the lemma names. If the analyst follows certain naming conventions this allows the proof to be automatically completed even without explicitly providing the hints.

Most of the practical experience with the strategies developed on this task was with the strategies aimed at proving a candidate component specification satisfied the requirements on components. Each component specified had 18 lemmas specified and one TCC generated that dealt with these requirements. Since 10 or so components were specified on the program, this gave around 200 lemmas with which to experiment. The set of theorems stated about compositions and refinements was much smaller so less experimentation was possible.

Generally, the strategy was found to be quite effective on the component restriction theorem classes. With minimal hints provided by the analyst, the strategy was able to automatically prove all but two of the approximately 200 tests run. Use of dependent typing in the specifications was the cause of the other two tests failing. Although the strategy could not complete the proofs automatically, it still made significant progress on the proofs.

For the other classes of theorems, too few examples were worked to make a meaningful assessment. For simple theorems such as showing that view relations are equivalence relations and showing that given witness states are elements of *init*, *css-prove* was found to quickly and automatically complete proofs in the examples tried. For more complicated theorems such as showing *impl_init(compose(set), cmp)* holds, the strategy was found to reduce the theorem to the desired point. Once the general approach for defining strategies was developed, strategies for new classes of theorems could usually be defined within an hour or two of time. This suggests that with a minimal amount of additional time the *css-prove* strategy could be extended to cover additional classes of theorems that arise when using the CSS framework.

In addition to the strategies themselves, we also documented instructions on how to write PVS proof strategies. The intent is that this information would provide a starting point for others who wish to modify our strategies or write their own.

8.4 Lessons

8.4.1 Specification Browser

The following lessons have been drawn from the Specification Browser work:

- **Configurability in the order of sections** — The requirement to allow configurability in the order of specification document sections increased the complexity of most features

of the Specification Browser. More work would have been completed on this first prototype if this requirement had been omitted.

- **Degree of control over specification document** — It is easy for a writer to bypass the Specification Browser and develop specification documents that do not follow the format produced by the browser. This would seriously undermine the ability of the tool to fulfill its function. Preventing specifiers from bypassing the Specification Browser would require a more sophisticated implementation than could be developed under this prototype effort. It would be necessary either to place the structural information (i.e., the formatted comments) maintained by the tool outside the text files themselves or to elevate them to commands that must be properly maintained by the specifier and parsed for the document to be considered well-formed input to the tool.
- **Low-level versus high-level view of structure** — The Specification Browser interface was initially conceived at a rather low level in which Emacs M-x commands were entered to instantiate template files and link them together. It was subsequently decided that a higher-level, outline-oriented interface would be more user friendly. The tool might have turned out better had we started thinking about the outline-oriented interface earlier.

8.4.2 Analyst's Assistant

The work resulted in the following observations:

- PVS strategies provide an effective mechanism for providing help to analysts. They are simple enough to write that the amount of time they save analysts justifies the investment to construct them.
- Strategies provide a useful bridge between how one would like to write specifications and how one is sometimes forced to write specifications. For example, consider the theorem class `impl_init(compose(-), -)`. The part of the proof addressed by `css-prove` is rewriting `init(compose(-))` as `init(cmp1) ∧ ... ∧ init(cmpn)`. This rewriting would not be necessary if the `compose` operator would explicitly set `init` to the conjunction of the `inits` for the individual components. This is not possible, though, since the number of components is unknown. Instead, a universally quantified expression is used to assert the result incorporates each component `init`. Because the proof strategy is executed at proof time when the number of components being composed is known, it can rewrite the expression in the more useful form.

As another example, the standard approach for writing CSS components is to first specify `spec_base`, then demonstrate `spec_base` satisfies the component restrictions, and finally specify `spec` to be equal to `spec_base`. This is necessary to allow PVS to successfully typecheck the theorems asserting the candidate component satisfies the component restrictions, but is annoying when proving other theorems since it is necessary to expand both `spec` and `spec_base` rather than simply expanding `spec`. The strategies can hide the additional layer of specification by automatically expanding `spec_base` whenever expanding `spec`.

- Due to the nature of this task, little thought was given to building blocks that would be of use in writing strategies. New functions were defined as needed to address the next class of theorem to be recognized by the strategy. Future strategy development would benefit from having a set of general purpose utilities upon which to build. For example, as mentioned earlier the `css-stewn` strategy could provide a powerful building block

for use in other strategies. In addition to simplifying the implementation of the existing strategies, this could simplify the construction of strategies in the future.

- The effectiveness of strategies is heavily dependent on the structure of the specifications. For example, some strategies that worked well on CSS specifications were found to work poorly on strategies written in a different form on another project. One way to view this lesson is that analysts should write their specifications in a standard format for which their proof strategies are tuned.²³ Another way to view this lesson is that the more specifications that can be used to test a strategy the better. Experience gained from additional testing might allow the strategy to be made more tolerant of differing specification styles.
- A trade-off must often be made between the number of steps automatically executed by the strategy and the complexity of the goals left for the analyst to prove. For example, each strategy could finish by attempting a grind, but for more complicated theorems this would take a long time and leave the analyst with a large number of goals to prove. Making this trade-off is somewhat subjective and was difficult to do on this effort due to the small number of examples worked.

8.5 Future Work

8.5.1 Specification Browser

The following are some of the areas for future work on the Specification Browser:

- **HTML output** — Post script viewers are not readily available on all computing platforms. Conversion of the \LaTeX to HTML is suggested because web browsers (HTML viewers) are available on most computing platforms at no cost. HTML output could allow people to participate in document reviews more easily.
- **Automatic File Creation** — The Specification Browser is not completely automatic when it comes to creating and linking parts of a specification document while viewing the *TLS OUTLINE* buffer, the Specification Browser outline interface. Three commands, `tls-add-comp-spec`, `tls-add-transition`, and `tls-add-processing-table`, only provide instructions on actions required to add parts to a specification document.

The problem of determining where to position new parts of a specification document was more difficult than expected. Consider creating and adding a component specification. If the *TLS OUTLINE* buffer is displaying a document outline only detailed to the level of showing other component specifications, then it is fairly easy, based on the cursor position, where the newly created component specification should be placed.

If the *TLS OUTLINE* buffer is displaying a document outline which shows all the details of a document outline, the cursor position does not always clearly identify the position for the new component specification. When the cursor is immediately before or after the start or end of a component specification, the position of the new component specification can be determined. A set of rules is needed to guide the Specification Browser in positioning a new component specification when the cursor is in a low-level section of an existing component specification

If `tls-add-comp-spec`, `tls-add-transition`, and `tls-add-processing-table` were modified to more fully automate the process of creating and linking parts of a specification document, the Specification Browser would be a more user-friendly tool.

²³ Users of the `css-prove` strategy should strive to write specifications in the form used for the CSS specifications.

- **Request Transition State Changes** — At several levels in a specification document a sequence of related sections is allowed. For example, the entire document can consist of a sequence of component specifications; each component specification can contain a sequence of request specifications; and each request specification can contain a sequence of transitions that describe the various processing cases. Due to lack of time this last example has not been fully integrated into the tool. If it were added it would be desirable to also implement functionality to check consistency between the transition sections included and the rows of the processing table.

8.5.2 Analyst's Assistant

Areas in which additional work could be done include:

- Building a larger set of specifications to use as a testbed. This would provide data regarding the robustness of the strategies.
- Constructing general purpose strategies that could be used as building blocks for other strategies. Essentially, this would result in a strategy development kit (SDK). Rewriting the existing strategies with such an SDK would make them more maintainable. In addition, future strategies could be constructed more easily.
- Improving how the `css-prove` strategy handles the currently recognized classes of theorems. For example, to prove component restrictions using `css-prove` currently requires the analyst provide hints. Some of these hints could be guessed at by the strategy. This would make the proofs slightly more automated.
- Expanding the set of classes recognized by the `css-prove` strategy. This would expand the scope of the help provided to analysts.

Section 9

Program Conclusions

This report has provided an overview of the work performed, accomplishments and lessons learned on the Composability for Secure Systems program. A PVS framework for composition and refinement reasoning has been developed. The application of this framework to the analysis of functional correctness, design refinement, fault tolerance and security has been demonstrated. We have also explored the use of tools to make this analysis easier and more cost effective. Questions remaining for future work are outlined in each of the foregoing sections.

Appendix A Bibliography

- [1] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. Technical Report 91, Digital Equipment Corporation, Systems Research Center, October 1992.
- [2] Martín Abadi and Leslie Lamport. Conjoining specifications. Technical Report 118, Digital Equipment Corporation, Systems Research Center, December 1993.
- [3] Ricky W. Butler and George B. Finelli. The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software. *IEEE Transactions on Software Engineering*, 19(1):3–12, January 1993.
- [4] Todd Fine. A Framework for Composition. In *Proceedings of the Eleventh Annual Conference on Computer Assurance*, pages 199–212, Gaithersburg, Maryland, June 1996.
- [5] Norman C. Hutchinson and Larry L. Peterson. The α -Kernel: An Architecture for Implementing Network Protocols. *IEEE Trans. Software Eng.*, 17(1):64–76, January 1991.
- [6] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
- [7] Daryl McCullough. Noninterference and the composability of security properties. In *Proceedings of the 1988 Symposium on Security and Privacy*, pages 177–186. IEEE, April 1988.
- [8] NCSC. Trusted Computer Systems Evaluation Criteria. Standard, DOD 5200.28-STD, US National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, December 1985.
- [9] S. Owre, N. Shankar, and J.M. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA 94025.
- [10] John Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical report, SRI International, December 1992.
- [11] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
- [12] Secure Computing Corporation. DTOS Composability Study. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, September 1995. DTOS CDRL A020.
- [13] Secure Computing Corporation. DTOS Formal Top-Level Specification (FTLS). Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, December 1996. DTOS CDRL A005.
- [14] Secure Computing Corporation. Composability for Secure Systems Refined Design Report. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, December 1997. CSS CDRL A006.

- [15] Secure Computing Corporation. Composability for Secure Systems Top Level Specification (TLS). Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1997. CSS CDRL A004.
- [16] Secure Computing Corporation. Assurance in the Fluke Microkernel: Formal Top-Level Specification. CDRL A004, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, 1998. In preparation.
- [17] Secure Computing Corporation. Assurance in the Fluke Microkernel: System Composition Study Report. CDRL A005, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, 1998. In preparation.
- [18] Secure Computing Corporation. Composability for Secure Systems Fault Tolerance Analysis Report. Technical report, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, February 1998. CSS CDRL A009.
- [19] Secure Computing Corporation. Composability for Secure Systems Final Report. Technical report, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1998. CSS CDRL A010.
- [20] Secure Computing Corporation. Composability for Secure Systems Modified Top-Level Specification. Technical report, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, (to appear) 1998. CSS CDRL A008.
- [21] Secure Computing Corporation. Composability for Secure Systems Security Policy Composability Results (SPCR). Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1998. CSS CDRL A005.
- [22] Secure Computing Corporation. Composability for Secure Systems Tools Report. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, February 1998. CSS CDRL A007.
- [23] Secure Computing Corporation. Formal Specifications, Hypervisors for Security and Robustness Program. CDRL A007, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, February 1998.
- [24] Secure Computing Corporation. Hypervisors for Security and Robustness: Final Report. CDRL A003, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, March 1998.
- [25] N. Shankar. A lazy approach to compositional verification. Technical Report TSL-93-08, SRI International, December 1993.

***MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)***

*The advancement and application of Information Systems Science
and Technology to meet Air Force unique requirements for
Information Dominance and its transition to aerospace systems to
meet Air Force needs.*